Informatik-Studium

Grundlagen der Informatik

Grundlagen der Informatik

Prof. Dr. Peter Baeumle-Courth
FHDW Fachhochschule der Wirtschaft

Prof. Dr. Peter Baeumle-Courth, FHDW Fachhochschule der Wirtschaft, Bergisch Gladbach
Die vorliegende Version wurde zuletzt aktualisiert am 14 September 2022

Vorwort

Dieses Skriptum basiert auf dem seit 1997 erstellten Skriptum „Wirtschaftsinformatik", das in den Versionen von 2000 bis 2002 vom Autor gemeinsam mit Prof. Stefan Nieland, FHDW Paderborn, ausgestaltet worden ist. Ihm gebührt daher Dank für die Mitwirkung bei einer Reihe von Abschnitten sowie für zahlreiche Korrekturvorschläge zum gesamten bisherigen Skriptum.

Gegenüber dem Skriptum „Wirtschaftsinformatik" kamen insbesondere die Abschnitte zu Algorithmen und Datenstrukturen, zur Kryptographie und zur Theoretischen Informatik hinzu. Daneben wurden eine Reihe von Details verbessert und für die Zielgruppe der angehenden Diplom- Wirtschaftsinformatiker/innen angepasst.

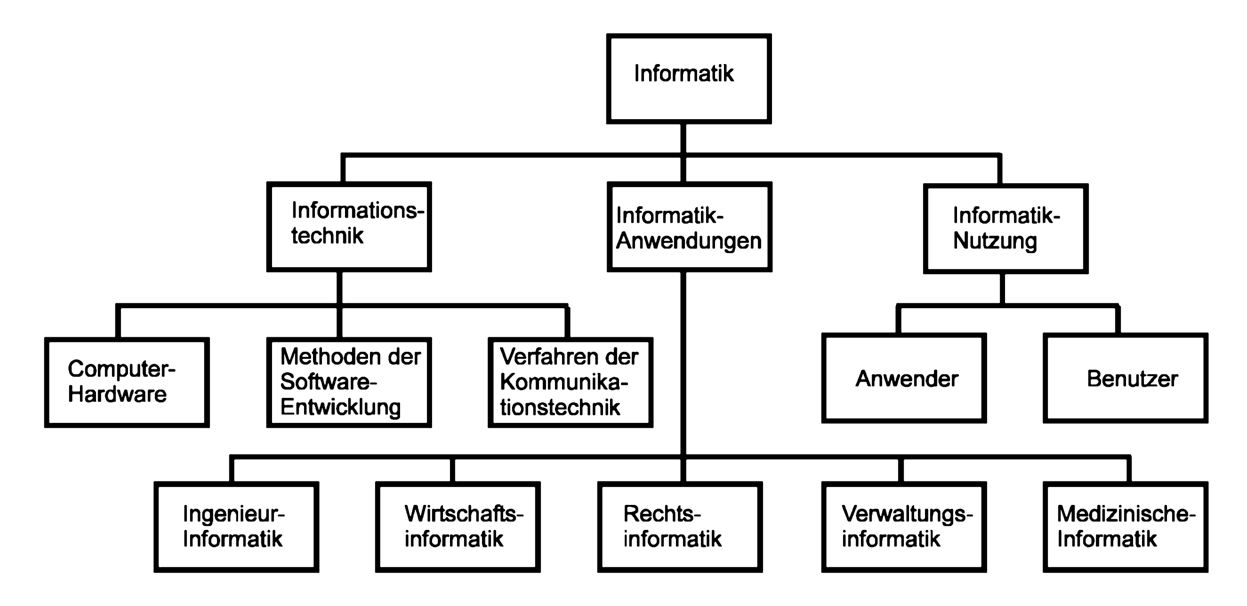
Im Februar 2004 erschienen einige Teile dieses Skriptums - ergänzt um Kapitel zu Analyse, Einführung und Einsatzgebieten von betrieblichen Informationssystemen - im Oldenbourg-Verlag als eigenständiges Buch "Wirtschaftsinformatik" gemeinsam mit Prof. S. Nieland (FHDW Paderborn) und Prof. H. Schröder (FH Nordakademie); sh. hierzu die entsprechende Literaturangabe [BaeNieSchr] auf S. 246.

Die hier vorliegende Fassung dieses Skriptums wurde im Juni 2004 um Teile zur Codierungstheorie und Datenkompression erweitert. Daneben wurden einige kleinere Modifikationen und Korrekturen am bisherigen Text vorgenommen. Speziell danke ich der Gruppe bfw403 für sachdienliche Korrekturhinweise zum Kapitel 5 (Kryptologie).

Im September 2022 wurde die vorliegende Fassung editiert.
Für weitere ergänzende Anmerkungen sowie Hinweise auf mögliche Fehler ist der Autor stets dankbar.

EINLEITUNG

Informatik[1] ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen. Man unterscheidet die Teilgebiete *Theoretische Informatik* (Automatentheorie, Schaltwerktheorie, Formale Sprachen (vgl. [Balke])), *Technische Informatik* (Schalttechnologie und Rechnerarchitekturen), *Praktische Informatik* (Betriebssysteme (vgl. [Bengel]), Compilerbau (vgl. [Aho] und [Baeumle2]), Softwaretechnologie) und *Wirtschaftsinformatik*.



Übersichtsgraphik entnommen aus [Stahlknecht].

Die *Gesellschaft für Informatik* (GI), die Standesvertretung der Informatiker, definiert Informatik als Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen; die Informatik umfasst

- die Informationstechnik (Computer-Hardware, Methoden der Softwareentwicklung, Verfahren der Kommunikationstechnik,
- die Informatikanwendungen in allen Fachgebieten und
- die Informatik-Nutzung durch Anwender und Benutzer.

Die Informatikanwendungen, zu denen die Wirtschaftsinformatik gehört, werden auch als „Bindestrich-Informatiken" bezeichnet.

Die in der obigen Abbildung aufgeführten Informatik-Anwendungen lassen sich durch Ihre Hauptarbeitsgebiete wie folgt charakterisieren.

- Ingenieurinformatik: Statik, Vermessungstechnik, Verkehrswesen und alle anderen Ingenieurswissenschaften; computergestütztes Konstruieren (CAD = Computer Aided Design), computergestützte Fertigung (CAM = Computer Aided Manufacturing) einschließlich numerisch gesteuerter Werkzeugmaschinen, Prozeßautomatisierung, Robotik.

---

[1] Eine sehr schöne, breit angelegte Einführung in die Informatik mittels konkreter Fragestellungen bietet das Buch „Der Turing Omnibus" [Dewdney]. Dabei gehen die Antworten des Buches allerdings davon aus, dass der Leser oder die Leserin nicht durch die eine oder andere mathematische Formel abgeschreckt wird.

- Wirtschaftsinformatik: die Umsetzung von einzelnen Anwendungen in ablauffähige, vernetzte Systeme, das Informationsmanagement (Bereitstellen von Werkzeugen der Informations- und Kommunikationstechnologie), Fragen der Beschaffung erforderlicher Soft- und Hardware, Aufwandsabschätzungen für die Datenorganisation, nicht zuletzt aber die Festlegung bzw. der Entwurf der relationalen Datenbankschemata und die Fragen der Informationsaufbereitung z.B. für die Unternehmensleitung. Hierfür haben wir ein eigenes Kapitel zum Thema „Relationale Datenbanken" vorgesehen (S. 198 ff).
- Rechtsinformatik: Juristische Informations- und Dokumentationssysteme, Datenschutzgesetzgebung, Vertragsgestaltung bei Hardware- und Softwarebeschaffung, Urheberschutz für Software, Computerkriminalität.
- Verwaltungsinformatik: Einwohnermeldewesen, Finanzverwaltung, Polizei, Haushaltswesen, Liegenschaftsverwaltung, Bevölkerungsstatistik.
- Medizinische Informatik: Befunderhebung und -auswertung, Therapieplanung, Labor-Analyse, Computer-Tomographie.

Dieses Skriptum stellt nur eine Ergänzung zur Vorlesung dar; es muss selbstverständlich nicht alles in der Veranstaltung selbst behandelt werden, was hier erwähnt wird. Auf der anderen Seite kann es jederzeit sein, dass kurzfristig weitere Inhalte in die Vorlesung aufgenommen werden, die zum Zeitpunkt der Fertigstellung noch nicht in diesen Text eingeflossen sind.

Ein über 800 Seiten umfassendes, m.E. sehr gutes Einführungswerk in zahlreiche Aspekte der Informatik ist der "Grundkurs Informatik" von Hartmut Ernst, das hiermit für einen weitergehenden Einstieg empfohlen wird; vgl. [Ernst] im Literaturverzeichnis.

1. DIE ERSTEN GRUNDLAGEN

Zunächst einmal sollen in ziemlich kompakter Form die Grundlagen eines elektronischen Datenverarbeitungssystems vorgestellt werden.

1.1. Historie und Grundbegriffe

1.1.1. Historie[2]

Die Grundlage für die Entwicklung zum Rechnen mit Maschinen bildet das in Indien um 500 n.Chr. entstandene und über die arabische Welt zu uns gelangte sogenannte *Hindu-Arabische Zahlensystem* mit den zehn Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9.

Im Vergleich zu dem aus Asterix-Heften bekannten *Römischen Zahlensystem* (z.B. MCMVII), das im 3. Jahrhundert v. C. aufkam, kennt das „Zehnersystem" die Zahl 0, und es ist ein Stellenwertsystem, das - wie unsere Erfahrungen zeigen - sehr viel einfacher handhabbar ist.

Im Jahre 1623 entwickelt der Theologe und Mathematiker Schickard für den Astronomen Kepler eine Rechenuhr, die auf dem Zählradprinzip aufbaute. Damit waren Addition und Subtraktion möglich, wobei mit sechs Stellen und einem Übertrag gearbeitet wurde.

1641 baut der französische Mathematiker Blaise Pascal mit 19 Jahren seinem Vater eine Addiermaschine mit sechs Stellen. 1650 wurde der Rechenschieber[3] erfunden. 1703 begann Gottfried Wilhelm Leibniz sich mit dem Dualsystem („Zweiersystem") zu beschäftigen, das zur Grundlage der heutigen Datenverarbeitung wurde.



1833 entwickelte der Mathematik-Professor Charles Babbage eine mechanische Rechenanlage „Difference Engine"; die Architektur dieser Anlage bestand aus einem Speicher (*store*), einem Rechenwerk (*mill*), einem Steuerwerk (*control*), einer Ein-/Ausgabeeinheit (*input/output*) und einem in Form von Lochkarten gespeicherten Programm.

1890 führte Hollerith im Rahmen der elften amerikanischen Volkszählung die Lochkartentechnik ein. Seit ca. 1920 wurden leistungsfähige Büro-Lochkartenmaschinen (z.B. von der Firma Bull) entwickelt.

1936 entwickelte der Bauingenieur Konrad Zuse seine auf dem Dezimalsystem basierende Rechenanlage Z1, welche die immer wiederkehrenden Berechnungen im Bereich der Statik automatisieren sollte. 1941 konstruierte er die Z3, die auf dem Dualsystem basierte, einen

---

2 Teilweise zitiert nach [Dworatschek].

3 Der Rechenschieber dürfte übrigens das bekannteste Beispiel eines Analog-Rechners sein, bei dem also nicht diskrete Werte, sondern ein stetiger Bereich von Werten verarbeitet wird. Anders formuliert: bei einem Rechenschieber kann man die Zahl 1 auch „ungenau" einstellen...

Relaisrechner mit Lochstreifenein- und -ausgabe, das ein Programm in Form eines Lochstreifens speichern konnte.



Abbildung: Lochstreifen
Quelle: https://de.wikipedia.org/wiki/Lochstreifen#/media/Datei:Lochstreifen-2.png

Erst mit dem Einzug von *Java*[4] seit ca. 1995 splittet sich der Bereich der Softwareentwicklung wieder auf, und es ist davon auszugehen, dass C/C++ und Java ihre jeweiligen Einsatzfelder finden und behalten werden.

Unter dem Begriff der „Mittleren Datentechnik" (MDT) wurden ab den Sechziger Jahren Rechner konstruiert, die die Gerätekomponenten aus der Büromaschinentechnik integriert hatten: Tastatureingabe, Formulartransport und Drucker. Dazu kam die verstärkte Nutzung der Magnetspeicherung.

Aus der Entwicklung spezialisierter Computer für die Steuerung und Überwachung technischer Prozesse entstand der Zweig der sogenannten „Mini-Computer". Hierzu gehören

[4] Vereinfacht ausgedrückt könnte man Java als „die Programmiersprache für das Internet" beschreiben.

die legendäre PDP 11 von Digital Equipment (siehe Bild rechts) oder die HP 3000 Serie von Hewlett Packard.



Seit 1971 sind „Mikro-Computer" auf dem Markt, die sich durch eine sehr hohe Integration der Schaltungstechnik auszeichnen. Aus diesem Sektor heraus erwuchsen dann seit 1981 auch die Home und Personal Computer heutiger Prägung, aber auch die Apple Macintosh.

1.1.2. Grundbegriffe

In der DIN 44 300 wird ein *Datenverarbeitungssystem* definiert als eine Funktionseinheit zur Verarbeitung von Daten, d.h. zur Durchführung mathematischer, umformender, übertragender und speichernder Operationen.

Daten stellen dabei Informationen (Angaben über Sachverhalte oder Vorgänge) aufgrund bekannter oder unterstellter Abmachungen in einer maschinell verarbeitbaren Form dar[5].

Unter *analoger* Rechentechnik versteht man das Verarbeiten kontinuierlicher Signale, z.B. Messung eines Zeigerausschlages (ideell gesehen: beliebig genau); demgegenüber sind bei *digitaler* Rechentechnik nur endlich viele, diskrete Werte möglich. Auf einer klassischen Analoguhr könnte man theoretisch der Position des Stundenzeigers alleine die genaue Uhrzeit entnehmen. Bei einer Digitaluhr werden dagegen nur ganzzahlige Werte von Minuten (oder ggf. Sekunden) angezeigt, eine genauere Information ist der digitalen Angabe nicht zu entnehmen.

---

[5] Zitiert nach [Hansen], S. 10.

1.2. Logik

In der formalen Logik[6] wird systematisch untersucht, wie man Aussagen miteinander verknüpfen und auf welche Weise man formale Schlüsse ziehen, Beweise durchführen kann.

In unserem Rahmen unterscheiden wir die *Aussagenlogik* und die *Prädikatenlogik*, die wir beide kurz vorstellen wollen.

1.2.1. Aussagenlogik

In der Aussagenlogik werden einfache Verknüpfungen mittels „und” und „oder” zwischen Aussagesätzen (oder deren Negation) untersucht. Beispiele für solche Aussagesätze sind

A := „Düsseldorf ist die Landeshauptstadt von Nordrhein-Westfalen”

B := „Die FHDW hat ihren Hauptsitz in Paderborn”

C := „Bergisch Gladbach liegt westlich von Köln”

Diese „atomaren” Aussagen können jeweils *wahr* (true) oder *falsch* (false) sein. Man spricht hierbei von einem *Wahrheitswert*.

Im Rahmen der Aussagenlogik interessiert man sich für die Zusammenhänge, wie sich aus den Wahrheitswerten solcher elementaren Aussagen die von komplexeren Zusammensetzungen ergeben wie beispielsweise:

D := (A und C) = „Düsseldorf ist die Landeshauptstadt von Nordrhein-Westfalen” und „Bergisch Gladbach liegt westlich von Köln”.

Die Aussage D ist, wie wir wissen, falsch, da bereits die Teil-Aussage C falsch ist.

Wir definieren: Symbole, die Aussagesätze oder kurz Aussagen repräsentieren, nennen wir *atomar* oder *Atome*. Jedem Atom können wir (eindeutig) einen der Wahrheitswerte wahr oder falsch zuordnen.

Folgende fünf logische Verknüpfungen werden in der Aussagenlogik üblicherweise betrachtet.

Symbol (Notation)	**Bedeutung**	**Bezeichnung**
$\neg$	nicht	Negation
$\wedge$	und	Konjunktion
$\vee$	oder	Disjunktion
$\Rightarrow$	wenn, dann	Implikation
$\Leftrightarrow$	genau dann, wenn	Äquivalenz

Anmerkung: formal käme man mit weniger als diesen fünf Operationen aus. Es ist jedoch sehr zweckmäßig, in der Praxis diese fünf Verknüpfungen einsetzen zu können.

[6] Zitiert nach [Schöning], S. 11.

Sehen wir uns einige Beispiele und allgemeinen Sachverhalte an.

1. Ist eine Aussage A wahr, so ist die Negation $\neg$ A (lies: „nicht A”) falsch.
   Wenn der Satz „Rom ist die Hauptstadt Italiens” wahr ist, dann ist die Aussage
   $\neg$ „Rom ist die Hauptstadt Italiens” = „Rom ist nicht die Hauptstadt Italiens”
   falsch.

2. Die verknüpfte Aussage A $\wedge$ B (lies: „A und B”, *Konjunktion*) ist genau dann wahr, wenn A wahr ist und B wahr ist.
   „Ich habe Urlaub” $\wedge$ „ich fahre nach Frankreich” (umgangssprachlich: „ich habe Urlaub und fahre nach Frankreich”) ist genau dann wahr, wenn beide Sachverhalte zutreffen.

3. Die Oder-Verknüpfung (Disjunktion) A $\vee$ B („A oder B”) ist genau dann wahr, wenn mindestens eine der beiden Aussagen A oder B zutreffen. Das heißt umgekehrt: A $\vee$ B ist nur dann falsch, wenn A falsch ist und B falsch ist.
   „Ich gehe in's Kino” oder „ich besuche ein Konzert” ist bereits dann wahr, wenn ich beispielsweise tatsächlich in ein Konzert gehe.
   „Ich gehe vormittags in die Vorlesung” oder „ich gehe nachmittags in's Schwimmbad” ist - entgegen dem üblichen Verständnis - aber auch dann wahr, wenn ich beides erfülle, also die Vorlesung am Vormittag besuche und am Nachmittag in das Schwimmbad gehe!

4. Die *Implikation* A $\Rightarrow$ B („wenn A, dann B”) ist so definiert, dass sie nur dann falsch ist, wenn A wahr ist, B jedoch falsch. Jede andere Konstellation führt zu einer wahren Implikation. Aus einer falschen (bzw. nicht erfüllten) Voraussetzung kann man im Sinne der Aussagenlogik alles schließen.
   Wahr sind also die folgenden Beispiele: „wenn der Papst eine Frau ist[7], dann regnet es jeden Tag in Rom”, „wenn die Temperatur im Zimmer des Direktors auf über 45 Grad ansteigt, dann bekommen die Schüler hitzefrei” sowie „wenn Elefanten acht Beine haben, dann essen Schnecken Mäuse”.
   Die Implikation wird also nur dann falsch, wenn wir aus einer wahren Aussage eine falsche schließen (wollen): „wenn Rom die Hauptstadt Italiens ist, dann ist der Papst eine Frau”.

5. Die *Äquivalenz* A $\Leftrightarrow$ B („A ist äquivalent zu B”) schließlich ist genau dann wahr, wenn A und B dieselben Wahrheitswerte besitzen.
   Wahr sind also die Äquivalenzen „wenn Rom die Hauptstadt Italiens ist, dann ist Paris die Hauptstadt von Frankreich” und „wenn Paris die Hauptstadt Italiens ist, dann ist Rom die Haupstadt von Frankreich”.
   Falsch ist eine Äquivalenz, wenn eine Seite falsch, die andere aber wahr ist: „wenn Rom die Hauptstadt Italiens ist, dann ist Rom die Hauptstadt von Frankreich”.

[7] Das Beispiel mit dem weiblichen Papst eignet sich in diesem Rahmen einigermaßen gut, weil nach bisheriger Erfahrung davon auszugehen ist, dass die katholische Kirche von ihren diesbezüglichen Gewohnheiten nicht gerade im Verlauf dieser Vorlesung abweichen wird.

Die formalen Definitionen dieser fünf Verknüpfungen (Operationen) lassen sich durch sogenannte *Wahrheitswertetafeln* darstellen.

A	**¬A**
w	f
f	w

In der obigen Wahrheitswertetafel wird die Negation dargestellt: ¬A ist falsch, wenn A wahr ist; ¬A ist wahr, wenn A falsch ist.

A	**B**	**A ∧ B**	**A ∨ B**	**A ⇒ B**	**A ⇔ B**
w	w	w	w	w	w
w	f	f	w	f	f
f	w	f	w	w	f
f	f	f	f	w	w

In der zweiten Wahrheitswertetafel werden die anderen Operationen, die zwei Aussagen miteinander verknüpfen, dargestellt. Man spricht hier von *binären* oder zweistelligen Verknüpfungen. (Die Negation ist demgegenüber eine *unäre* oder einstellige Operation.)

Während die Definitionen der ersten drei Verknüpfungen einem normalen Menschen nachvollziehbar sein dürften, sträubt sich der gesunde Menschenverstand zunächst gegen die Festlegung der Implikation und der Äquivalenz.

Jedoch sind die obigen Definitionen vielleicht besser verständlich, wenn wir uns die folgenden üblichen Schlußfolgerungen ansehen.

Zur Implikation: gelten die beiden Aussagen A ⇒ B und ¬ A ⇒ B, dann ist B offensichtlich unabhängig von A (bzw. ¬ A) erfüllt.
Formelhaft könnte man also schreiben[8]: (A ⇒ B) ∧ (¬ A ⇒ B) ⇒ B.

Zur Äquivalenz: die „genau dann, wenn"-Verknüpfung wird üblicherweise als „wenn A wahr ist, dann ist auch B wahr - und umgekehrt" wiedergegeben. Das bedeutet aber auch, dass A falsch sein muss, wenn B falsch ist.

Man spricht von einer *Tautologie*, wenn die Aussage immer wahr ist. Ein elementares Beispiel einer Tautologie ist der Ausdruck[9]: A ⇒ A.

---

[8] Der Autor ist sich bewußt, dass diese Formel bei mathematisch nicht besonders ambitionierten Leserinnen und Lesern ein gewisses Unbehagen auslösen kann. Es ist allerdings auf dem Gebiet der EDV leider unvermeidbar, gelegentlich auf abstrakte Zusammenhänge und deren formale Schreibweisen zu stoßen.

[9] Dies ist übrigens ein weiterer Grund, weshalb bei der Implikation aus etwas Falschem alles gefolgert werden kann: A ⇒ A ist sicherlich immer wahr, und dies soll auch so sein, wenn A selbst falsch ist!

Mit Hilfe der Wahrheitswertetafeln kann man sehr rasch Zusammenhänge erkennen. Beispielsweise gilt, dass die Formeln $A \Rightarrow B$ und $\neg A \vee B$ äquivalent sind.

A	**B**	**$A \Rightarrow B$**	**$\neg A \vee B$**
w	w	w	w
w	f	f	f
f	w	w	w
f	f	w	w

Einige allgemein gültige Gesetze der Aussagenlogik seien nachfolgend aufgelistet, ohne dass diese hier formal bewiesen werden sollen.

Kommutativgesetze	$A \wedge B = B \wedge A$ $A \vee B = B \vee A$
Assoziativgesetze	$(A \wedge B) \wedge C = A \wedge (B \wedge C)$ $(A \vee B) \vee C = A \vee (B \vee C)$
Distributivgesetze	$(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$ $(A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C)$
Doppelte Negation hebt sich auf	$\neg (\neg A) = A$
Gesetze von DeMorgan	$\neg(A \wedge B) = \neg A \vee \neg B$ $\neg(A \vee B) = \neg A \wedge \neg B$

1.2.2. Prädikatenlogik

Es gibt allerdings auch Sachverhalte, die nicht mit den einfachen Mitteln der Aussagenlogik beschrieben werden können.

Ein Beispiel[10] hierzu: Alle Unternehmer wollen Gewinne machen. Weil Herr Schmidt ein Unternehmer ist, will er Gewinne machen. Diese Gedankenkette ist intuitiv sicherlich korrekt.

Formuliert man aber im Sinne der Aussagenlogik
G := „Alle Unternehmer wollen Gewinne machen"
U := „Herr Schmidt ist ein Unternehmer"
S := „Herr Schmidt will Gewinne machen"

Dann ist S im Sinne der Aussagenlogik keine Konsequenz aus G und U, denn wir können zeigen, dass $G \wedge U \Rightarrow S$ nicht allgemeingültig ist.

[10] Zitiert nach [Beedgen], S. 89.

G	U	S	$G \wedge U$	$G \wedge U \Rightarrow S$
w	w	w	w	w
w	**w**	**f**	**w**	**f**
w	f	w	f	w
w	f	f	f	w
f	w	w	f	w
f	w	f	f	w
f	f	w	f	w
f	f	f	f	w

Die Wahrheitswertetafel zeigt (in der grau unterlegten Zeile), dass $G \wedge U \Rightarrow S$ keine Tautologie, also nicht immer wahr ist.

Im Folgenden soll kurz in die sogenannte *Prädikatenlogik* eingeführt werden, mit der solche Fragestellungen wie die im obigen Beispiel behandelt werden können. Die Prädikatenlogik ist gleichzeitig wesentliche Grundlage für die *logische Programmierung*, deren bekannteste Programmiersprache *ProLog* (Programmieren in Logik) sein dürfte.

Die Theorie der Prädikatenlogik wurde bereits zu Beginn des 20.Jahrhunderts gelegt; in der Informatik eingesetzt wird sie seit den Sechziger Jahren.

Beispiele:

1. Wir bringen „a ist teurer als 3" zum Ausdruck. Dazu definieren wir uns ein sogenanntes *Prädikat* TEURER(a,b) mit der Bedeutung „a ist teurer als b". [Ein solches Prädikat ist gleichzeitig eine *Relation*. Vgl. hierzu das Kapitel zu relationalen Datenbanken (siehe S. 198).]
2. Entsprechend kann man die Relation bzw. das Prädikat LIEFERT(a,b) einführen: LIEFERT(„Fa. Meier","Kühlschrank") steht dann für „Fa. Meier liefert (einen) Kühlschrank".

Wir wollen einige fundamentale Begriffe der Prädikatenlogik formal definieren.

Ein *Term* kann wie folgt gebildet werden:

1. Eine *Konstante* ist ein Term. (Zum Beispiel: 5 oder „Kühlschrank".)
2. Eine *Variable* ist ein Term. (Eine Variable ist ein Platzhalter wie z.B. a.)
3. Ist f eine Funktion mit n Argumenten, eine sogenannte n-stellige Funktion, und sind a_1, a_2, ..., a_n Terme, dann ist $f(a_1,a_2,...,a_n)$ ein Term.
   (LIEFERT() ist eine zweistellige Funktion, sind a und b Variable und damit auch Terme, so ist LIEFERT(a,b) wiederum ein Term.)

Ein *Prädikat* P ist eine Abbildung (Funktion), die einem n-Tupel von Konstanten genau einen der Wahrheitswerte w(ahr) oder f(alsch) zuordnet.

Gemäß dieser Definition ist TEURER() ein Prädikat, denn je zwei konkreten Dingen (Konstanten) wird eindeutig w oder f zugeordnet: TEURER(„Auto", „PC") ist entweder wahr oder falsch, sofern „Auto" und „PC" für zwei konkrete Dinge (Konstanten) stehen.

Ausdrücke werden wie in der Aussagenlogik verknüpft; hinzu kommen aber zwei weitere Symbole (*Quantoren*) zur Spezifizierung von Variablen, der *Allquantor* ∀ und der *Existenzquantor* ∃. Mit diesen können Aussagen wie „Alle ... sind ..." oder „Es gibt ..." formalisiert werden, z.B. lässt sich „Alle Unternehmer wollen Gewinne machen" formalisieren in der Form ∀x: (x ist Unternehmer ⇒ x will Gewinn machen). Entsprechend ist „Es gibt PCs, die sind teurer als DM 5000" formal zu schreiben als ∃x: (x ist PC ∧ TEURER(x,5000)).

Wendet man die allgemeine Aussageform

∀x: (x ist Unternehmer ⇒ x will Gewinn machen)

an auf einen konkreten Unternehmer, erhält man eine konkrete Aussage: „Schmidt ist Unternehmer" ⇒ „Schmidt will Gewinn machen".

1.3. Zahlensysteme

Die Daten (bzw. Informationen), die bei einer DV-Anlage verwaltet werden müssen, werden digital abgespeichert, das heißt präziser: als eine Folge von Elementen, die jeweils 0 oder 1, wahr oder falsch, Strom leitend oder nicht leitend (usw.) sind; diese binäre Abspeicherung kann man sich vereinfacht als Folge von Nullen und Einsen vorstellen, ohne dass man sich um die physikalischen Gegebenheiten dieser Abspeicherungen kümmern müsste.

Natürlich setzen sich komplexere Informationen aus einer Vielzahl solcher 0-1-Bausteine, *Bits* genannt, zusammen. Zur Systematisierung dieser Informationen werden sogenannte *Codes* verwendet, die es in einheitlich genormter Weise erlauben, u.a. das gesamte Alphabet abzuspeichern. Zum späteren besseren Verständnis dieser Codes machen wir zunächst einen Ausflug in die Welt der Zahlensysteme

Eine einfache Ja/Nein-Entscheidung kann offenbar durch genau ein Bit abgespeichert werden, indem z.B. festgelegt wird, dass 1 für „Ja" und 0 für „Nein" stehen soll. Hier spricht man vom Informationsgehalt 1 Bit.

Nimmt man zwei Bit Speicherplatz her, so können offenbar gerade die vier Kombinationen 00, 01, 10 und 11 DV-technisch abgespeichert werden, d.h. mit 2 Bit können vier verschiedene Zustände dargestellt werden. Entsprechend dienen drei Bit zur Abspeicherung von acht ($=2^3$) verschiedenen Möglichkeiten, die schematisch dargestellt werden können als 000, 001, 010, 011, 100, 101, 110 und 111.

In naheliegender Weise wird man diese acht 0-1-Tripel (häufig) als 0, 1, 2, 3, 4, 5, 6 und 7 interpretieren. Dabei spricht man vom Zweiersystem (*Dualsystem*); ähnlich dem gewohnten Zehner- oder Dezimalsystem handelt es sich dabei um ein Stellenwertsystem: während beim Zehnersystem die Potenzen von 10 durch eigene Stellen dargestellt werden (1, 10, 100, 1000 usw.), sind es beim Zweiersystem die Potenzen von 2 (1, 2, 4, 8, 16 ...). Die duale Zahl 110 steht auf diese Weise für $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 1 = 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 6$ dezimal. Weitere Rechenübungen können auf Wunsch durchgeführt werden.

Es dürfte unmittelbar einleuchten, dass aufgrund der binären Abspeicherungsmethodik in der Datenverarbeitung das Dualsystem das dem Problemkreis angemessene Zahlensystem ist. In Kreisen hartgesottener Programmierer werden jeweils 4 Bit (16 Möglichkeiten) zu Einheiten

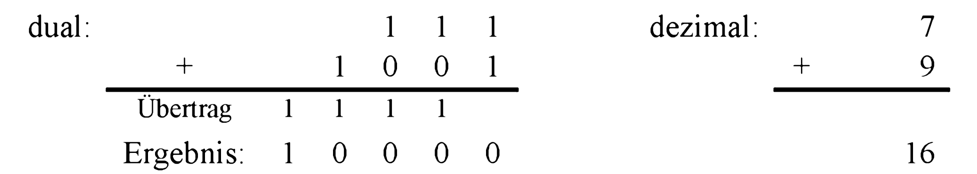
zusammengefasst, die mit dem *Hexadezimalsystem* (Sechzehnersystem) gut beschreibbar sind.

Soll beispielsweise eine Schulnote gespeichert werden, so sind drei Bit notwendig, da mit nur zwei Bit lediglich vier verschiedene Alternativen dargestellt werden könnten; in gewissem Sinn werden hierbei allerdings (zwangsläufig) zwei Möglichkeiten „verschenkt".

Um das Alphabet der deutschen Sprache mit seinen je 26 Klein- und Großbuchstaben, je 3 großen und kleinen Umlauten (usw.) sowie einer ganzen Reihe von Sonderzeichen (wie Klammern und Satzzeichen) abspeichern zu können, werden somit mindestens 6 oder 7 Bit benötigt (2^6=64, 2^7=128). Als Standard hat es sich durchgesetzt, dass 8 Bit, d.h. die Möglichkeit, 256 Alternativen abspeichern zu können, als ein *Byte* bezeichnet werden.



Das Rechnen in anderen Stellenwertsystemen als dem uns gewohnten Zehnersystem ist generell genauso schwer oder leicht, nur natürlich nicht so gewohnt. Eine Addition im Zweiersystem funktioniert nach dem gleichen Ablaufprinzip, das wir in unserer Kindheit einst für das Zehnersystem lernen durften.



So, wie man es sich bei Zucker und Mehl angewöhnt hat, 1000 Gramm als ein Kilogramm zu bezeichnen, werden in der DV-Welt 1024 Byte als ein *KiloByte* (kurz KByte, KB oder K) bezeichnet. Der zunächst etwas merkwürdige Wert 1024 ergibt sich aus dem Zweiersystem: $1024 = 2^{10}$.

Dennoch ist es natürlich umgangssprachlich legitim, bei einem KiloByte von (etwa) „tausend Bytes" zu sprechen. Ein Vergleich am Rande: eine Schreibmaschinenseite Text wird, grob gemittelt, häufig mit 2000 Bytes ≈ 2 K angesetzt[11].

Konsequent geht es dann weiter: 1024 KiloByte werden als ein *MegaByte* (kurz MB), 1024 MegaByte (1.048.576 KiloByte) als ein *GigaByte* (GB) bezeichnet.

1.4. Einführung in die Codierungstheorie

Für die Speicherung und Übertragung von Daten (Nachrichten) ist es bei Nutzung der Digitaltechnik wichtig, eine entsprechende technische Darstellung - eine sog. Codierung - zu wählen. Anders als bei analogen Techniken können hier nicht "beliebig viele" Zwischenwerte gespeichert, jeder Zustand muss eindeutig beschrieben ("codiert") werden.

[11] Hierbei wird allerdings nur der Informationsgehalt des puren Textes berücksichtigt. Moderne Textverarbeitungsprogramme wie Lotus Word Pro, StarWriter oder Corel WordPerfect speichern in der Regel wesentlich mehr Informationen, weshalb die abgespeicherten Textdateien wesentlich größer sind. Vgl. hierzu die Betrachtung auf Seite 39.

In diesem Kapitel wollen wir einige grundlegende theoretische Aspekte aus der Codierungstheorie behandeln, im darauffolgenden Kapitel wird dann eine Reihe konkreter Anwendungen diskutiert werden.

1.4.1. Definitionen zur Codierung

Unter einer Codierung versteht man mathematisch betrachtet eine umkehrbare Zuordnung aus einer Grundmenge G in eine Zielmenge Z. Dabei steht G in unserem Kontext für die "eigentlich interessante" Menge von Informationen, Daten, Nachrichten, während Z die DV-konforme Abspeicherungsform darstellt.

Die in der Praxis der Datenverarbeitung auftretenden Mengen G und Z bestehen aus allen möglichen (endlichen) Sequenzen von Zeichen aus bestimmten (endlichen) Alphabeten A und B. Mit A^* werden dann alle endlichen Folgen von Zeichen aus der Menge A gekennzeichnet[12]. Wir formulieren deshalb die nachfolgende Definition bereits in dieser etwas engeren Form (und nicht für beliebige Mengen G und Z).

1.4.1.1. Definition (Codierung, Decodierung, Code)

a) Es seien A ein endliches Ausgangs- und B ein endliches Ziel-Alphabet gegeben. Dann heißt eine umkehrbare (injektive) Abbildung $c: A^* \rightarrow B^*$ eine *Codierung*.

b) Die Menge $c(A^*)$ wird *Code* oder *Code-Menge* genannt.

c) Ist $s = a_1 a_2 ... a_n \in A^*$, so wird $c(s) = c(a_1)c(a_2)...c(a_n) \in B^*$ definiert. Für eine solche endliche Folge s von Elementen aus A wird c(s) als *Codewort* bezeichnet.

d) Die lokale Umkehrabbildung $d: c(A^*) \rightarrow A^*$ mit $d(c(a)) = a \ \forall a \in A$ wird *Decodierung* genannt.

e) Eine Codierung wird *homomorph* genannt, wenn sich die Codierung eines Wortes (also einer Sequenz von Zeichen) als Sequenz der Codierung der einzelnen Zeichen ergibt.

Mit einer Codierung wird also jedes Wort, d.h. jede endliche Folge von Elementen aus A, auf genau eine endliche Sequenz von Elementen aus B abgebildet; es muss jedoch nicht jede endliche Folge von B-Elementen auch vorkommen; mathematisch: die Codierungsvorschrift muss zwar injektiv, nicht aber surjektiv bzw. bijektiv sein[13].

Die Forderung der Homomorphie besagt, dass wir zeichenweise codieren können. Dies erscheint zunächst als absolut selbstverständliche Forderung, wir werden jedoch später sehen, dass es auch gute Gründe geben kann, davon abzuweichen.

---

12 Wer sich für diese Begrifflichkeiten aus formaler Sicht interessiert, findet entsprechende Definitionen im Kapitel zur Theoretischen Informatik; in 6.2.1.1. (S. 127) werden die Begriffe "Alphabet" und "Menge aller Worte" präzise definiert.

13 *Surjektiv* bedeutet, dass alle Elemente der Zielmenge auch bei der Abbildungsvorschrift vorkommen; *bijektiv* heißt eine Abbildung, wenn sie injektiv und surjektiv ist.

1.4.1.2. Beispiel (Morse-Code)

Beim bekannten Beispiel des Morse-Codes[14] wird das Alphabet A := { a, ..., z } auf Sequenzen von Punkten und Strichen abgebildet, also $Z = B^*$ mit B := { '.' , '-' }.

Beispielsweise wird das Zeichen *i* auf die Sequenz ". ." abgebildet, das Zeichen *s* wird durch ". . ." dargestellt. Damit folgt: c(iii) = c(ss), so dass keine Umkehrbarkeit auf Wortebene vorliegt! (Zeichenweise ist diese natürlich gegeben.)

Der Morse-Code wird also formal dahingehend erweitert, dass Pausen eingelegt werden, in das Zielalphabet also ein drittes Zeichen (Leerzeichen) mit aufgenommen wird. Dann ist die Menge B := { '.' , '-', ' ' } und c(iii) = ".. .. .." ist von c(ss) = "... ..." unterscheidbar[15].

Mit der sogenannten Fano-Bedingung wird die wortweise Invertierbarkeit sichergestellt.

1.4.1.3. Fano-Bedingung

Ist kein Codewort Anfangswort eines anderen Codewortes, dann kann jede Zeichenkette aus $c(A^*)$ auch dekodiert werden.

Weitere Beispiele für konkrete Codes werden wir im nächsten Kapitel kennenlernen; erwähnt sei an dieser Stelle bereits der Code "ASCII" zur Darstellung von Zeichen auf einem heutigen digitalen Rechnersystem (vgl. S. 36).

1.4.2. Redundanz von Codes

Codierungen von einzelnen Zeichen können auch mehrere Zeichen (des Zielalphabets) umfassen, dies haben wir im obigen Beispiel des Morse-Codes bereits gesehen.

Allgemein spielt die sogenannte mittlere Wortlänge eine Rolle.

1.4.2.1. Definition (mittlere Wortlänge)

Gegeben seien ein Ausgangsalphabet A, ein Zielalphabet B und eine homomorphe Codierungsvorschrift $c: A^* \to B^*$. Als *mittlere Wortlänge* des Codes wird definiert: $L := \sum p_i l_i$, wobei l_i die Wortlänge des codierten (i-ten) Zeichens ist und p_i die zugehörige Auftrittswahrscheinlichkeit (oder relative Häufigkeit). Die Summation läuft über alle Zeichen des Alphabetes A.

---

[14] Wir vereinfachen hier die Darstellung ein wenig.

[15] Wir wollen an dieser Stelle den mathematischen Formalismus nicht weiter diskutieren, denn der aufmerksame Leser oder die aufmerksame Leserin wird bemerkt haben, dass die formale Beschreibung modifiziert werden müsste!

1.4.2.2. Beispiel (mittlere Wortlänge)

Es seien $A := \{ w, x, y, z \}$ und $B := \{ 0, 1 \}$. Die Abbildungsvorschrift c werde wie folgt definiert.

$c(w) := 00,\ c(x) := 01,\ c(y) := 10, c(z) := 11.$

Die Auftrittswahrscheinlichkeiten der vier Elemente von A seien alle gleich und somit 0.25.

Dann berechnet sich die mittlere Wortlänge dieses Codes zu:

$L := \sum p_i l_i = 0.25 \cdot 2 + 0.25 \cdot 2 + 0.25 \cdot 2 + 0.25 \cdot 2 = 2.$

Das ist natürlich keine besondere Überraschung, denn alle Codierungssequenzen haben die Länge 2.

Modifizieren wir die Vorschrift und betrachten die Kodierung k_1 wie folgt.

$k_1(w) := 0,\ k_1(x) := 10,\ k_1(y) := 101, k_1(z) := 111.$

Dann berechnet sich die mittlere Wortlänge wie folgt:

$L := \sum p_i l_i = 0.25 \cdot 1 + 0.25 \cdot 2 + 0.25 \cdot 3 + 0.25 \cdot 3 = 2.25.$

Die so gewählte Codierungsvorschrift erfüllt jedoch nicht die Fano-Bedingung. (Warum?)

Daher wollen wir eine weitere Beispielcodierung betrachten: k_2 definiert durch

$k_2(w) := 0,\ k_2(x) := 10,\ k_2(y) := 110, k_2(z) := 111.$

Sind wiederum alle Auftrittswahrscheinlichkeiten innerhalb der Menge A gleich, so ergibt sich auch hier die mittlere Wortlänge von 2.25.

1.4.2.3 Information und Entropie

Aus der Informationstheorie[16] ist der Begriff der *Information* bekannt. Für den sogenannten Informationsgehalt[17] J eines Zeichens[18] wird gefordert:

a) Je seltener ein Zeichen auftritt, desto größer ist der Informationswert des Zeichens. Also: desto kleiner die Auftrittswahrscheinlichkeit p des Zeichens ist, desto höher ist die Information J.

b) Die Information einer Zeichenkette soll sich aus der Summe der Einzelinformationen ergeben: $J(a_1 a_2 ... a_n) = J(a_1) + J(a_2) + ... + J(a_n)$.

c) Spezialfall: wäre die Auftrittswahrscheinlichkeit eines Zeichens 1, dann soll die Information 0 sein.

---

16 Vgl. hierzu etwa [Ernst], S. 54 ff.

17 Wir schreiben der besseren Lesbarkeit wegen J statt des ansonsten üblichen I.

18 Analog lässt sich dies auch für Worte - also Zeichenketten - definieren.

Die mathematisch einfachste (übliche) Funktion, die diese Eigenschaften besitzt, ist der Logarithmus. Daher wird als Informationsgehalt (oder kurz Information) eines Zeichens definiert[19]:

$$J(a) := \log_2(\tfrac{1}{p(a)}) = ld(\tfrac{1}{p(a)}) = -ld(p(a)).$$

Als *Entropie H* einer Nachricht - also einer endlichen Sequenz von Zeichen - wird der mittlere Informationsgehalt definiert:

$$H(a_1 a_2 \ldots a_n) := \sum p(a_i) \cdot J(a_i) = \sum p(a_i) \cdot ld(\tfrac{1}{p(a_i)}),$$ Summation über *i* von *1* bis *n*.

Maximal wird die Entropie dann, wenn alle Auftrittswahrscheinlichkeiten im Ausgangsalphabet gleich sind. In diesem Falle ist $H = \sum \frac{1}{n} \cdot ld(n) = ld(n)$.

Generell gilt gemäß dem Shannonschen Codierungstheorem stets $H \leq L$.

Das bedeutet für die Praxis: ist die Entropie gleich der mittleren Wortlänge des zugrundeliegenden Codes, so kann bei der Codierung nichts "verbessert" werden im Sinne einer eventuellen kürzeren Darstellung. Ist jedoch die Entropie kleiner, so kann eine kompaktere Codierung gefunden werden, sofern man nicht eine zeichenweise Codierung erzwingt. Dies ist der Ansatzpunkt für Kompressionsverfahren und führt zur Definition der Redundanz.

1.4.2.4. Definition (Redundanz)

Für einen gegebenen Code wird dessen *Redundanz R* definiert durch $R := L - H$, also als Differenz von mittlerer Wortlänge und Entropie.

1.4.2.5. Beispiel (Redundanz)

Gegeben sei das Alphabet $A := \{ x, y, z \}$. Codiert werde wieder in Form von Bitfolgen, das Alphabet B bestehe also wieder aus 0 und 1. Dann ist ein konkretes Beispiel einer Codierung:

$$c(x) := 1, c(y) := 01, c(z) := 00.$$

Die Auftrittswahrscheinlichkeiten oder -häufigkeiten der drei Zeichen seien $\frac{1}{2}$, $\frac{1}{4}$ und $\frac{1}{4}$. Dieser Code hat daher eine mittlere Wortlänge von $L = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{4} \cdot 2 = \frac{3}{2} = 1.5$.

Die Entropie (bezogen auf ein Wort, bei dem jedes Zeichen genau einmal vorkommt, also etwa "xyz") berechnet sich zu:

$$H(xyz) = p(x) \cdot J(x) + p(y) \cdot J(y) + p(z) \cdot J(z) = \tfrac{1}{2} \cdot ld(2) + \tfrac{1}{4} ld(4) + \tfrac{1}{4} ld(4) = \tfrac{1}{2} + \tfrac{1}{2} + \tfrac{1}{2} = \tfrac{3}{2} = 1.5.$$

Das heißt die Redundanz in dieser Situation ist $R = L - H = 0$.

---

[19] *ld* steht für logarithmus dualis, den Logarithmus zur Basis 2. Dieser wird verwendet, weil man üblicherweise möchte, dass der Informationsgehalt bei dem "digitalen Grundszenario" von zwei möglichen Zeichen (0 und 1) mit gleicher Wahrscheinlichkeit (=0.5) für ein Zeichen gerade 1 sein soll. Ansonsten wäre aber auch jede andere Basis für die Logarithmusbildung möglich.

Wären die Auftrittswahrscheinlichkeiten gleich, also $\frac{1}{3}$ für jedes Zeichen, dann ergäbe sich folgendes Bild:

Mittlere Wortlänge: $L = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 2 = \frac{5}{3} \approx 1.66$,
Entropie: $H(xyz) = p(x) \cdot J(x) + p(y) \cdot J(y) + p(z) \cdot J(z) = ld(3) \approx 1.59$,
Redundanz: $R = L - H \approx 0.07$.

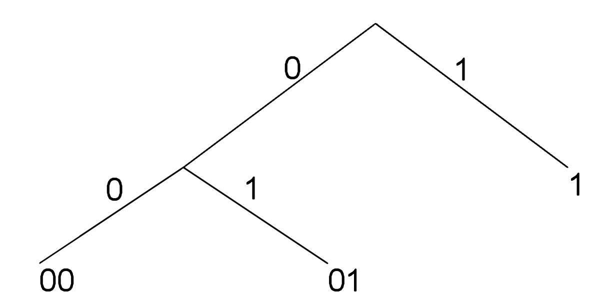
Dies bedeutet: die o.g. Codierung ist nicht ganz redundanzfrei, dies kann aber auch nicht auf der Ebene einzelner Zeichen bei Verwendung eines zwei-elementigen Zielalphabetes erreicht werden. Entweder nutzt man einen (z.B.) drei-elementiges Zielalphabet oder man codiert ganze Zeichenketten, wenn man die Redundanz auf 0 bringen möchte.

Generell versucht man zunächst einmal, Redundanz wo möglich zu vermeiden; umgekehrt dient diese aber zur Erhöhung von Übertragungs- oder Speicherungssicherheit, da mit geeigneten Codes bestimmte Fehler erkannt oder sogar korrigiert werden können. Darauf gehen wir nachfolgend etwas näher ein.

1.4.2.6. Code-Bäume

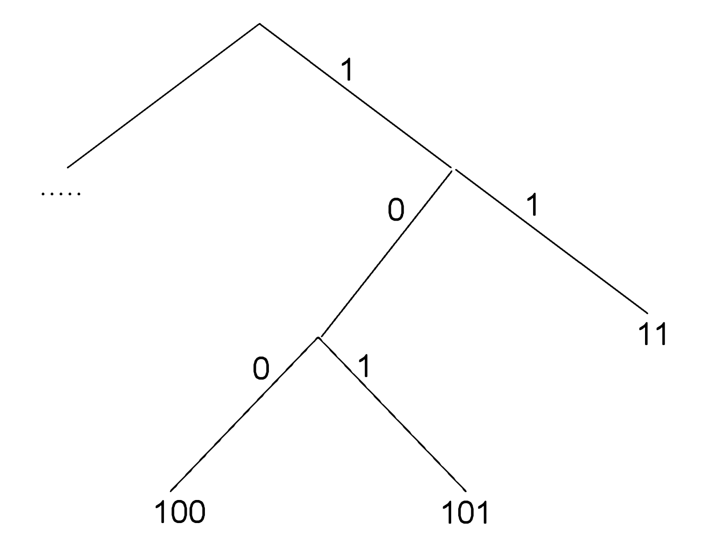
Um eine auftretende Code-Redundanz zu vermindern können Codes mit variabler Wortlänge aufgebaut werden, womit auf die unterschiedliche Verteilung der Ausgangszeichen Rücksicht genommen werden kann.

Hilfsmittel zur Betrachtung solcher Codes sind sog. *Code-Bäume*. Greifen wir auf obiges Beispiel zurück $[c(x) := 1, c(y) := 01, c(z) := 00]$, dann können wir dies graphisch wie folgt darstellen:



Ein Code ist gut geeignet (d.h. mit möglichst geringer Redundanz), wenn der zugehörige Code-Baum keine ungenutzten Blätter besitzt (wie es hier der Fall ist) und die kürzeren Wege stets für Worte mit höherer Auftrittswahrscheinlichkeit genutzt werden. Das bedeutet hier im Beispiel: die relative Auftrittshäufigkeit der Codierung “1” darf nicht kleiner sein als die von “00” oder “01”; wäre dies so, dann wäre ein Vertauschen der beiden Codierungen sinnvoll.

Betrachten wir dazu die folgende Codierung k - der Einfachheit halber wieder über dem kleinen Alphabet $A := \{ x, y, z \}$. Es sei: $k(x) := 11, k(y) := 101, k(z) := 100$. Dann sieht der zugehörige Code-Baum wie folgt aus, wobei der nicht genutzte Teil links nur angedeutet ist.



Seien die Auftrittswahrscheinlichkeiten wieder $\frac{1}{2}$, $\frac{1}{4}$ und $\frac{1}{4}$ für x, y und z bzw. 11, 100 und 101. Dann hat der Code eine mittlere Wortlänge von $L = \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 + \frac{1}{4} \cdot 3 = \frac{5}{2} = 2.5$.

Die Entropie berechnet sich natürlich wiederum zu

$$H(xyz) = p(x) \cdot J(x) + p(y) \cdot J(y) + p(z) \cdot J(z) = \frac{1}{2} \cdot ld(2) + \frac{1}{4} ld(4) + \frac{1}{4} ld(4) = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = \frac{3}{2} = 1.5.$$

Das heißt die Redundanz in dieser Situation ist $R = L - H = 1$. Offensichtlich ist diese Codierung nicht optimal, durch ein Nutzen des linken Teilbaumes ("Beginn mit dem ersten Zeichen 0") kann - wie oben dargestellt - die Redundanz auf 0 gesenkt werden.

1.4.2.7. Huffman-Algorithmus

Auf der Suche nach einem Verfahren, wie ein optimaler Code-Baum aufgestellt und somit die Redundanz minimiert werden kann, gibt der Huffman-Algorithmus die Antwort.

Alle auftretenden Zeichen werden nach aufsteigender Wahrscheinlichkeit sortiert. Die beiden Zeichen mit den kleinsten Wahrscheinlichkeiten p_1 und p_2 werden zu einem Knoten zusammengefasst, der mit der Wahrscheinlichkeit $p_1 + p_2$ bewertet wird. Damit erhält man eine neue Sequenz von Wahrscheinlichkeiten (für Zeichen bzw. Knoten), die man auf dieselbe Weise behandelt, bis schließlich nur noch ein Knoten übrig ist. Es ist erwiesen, dass es keinen günstigeren Code gibt, der auf der Codierung von Einzelzeichen beruht, als der auf dem so ermittelten Huffman-Baum beruhende Code.

Sehen wir uns dies in einem kleinen konkreten Beispiel an.

Betrachten wir das Alphabet $A := \{ a, b, c, d \}$ mit den Auftrittswahrscheinlichkeiten 0.4, 0.3, 0.2 und 0.1. Dann berechnet sich die Entropie zu

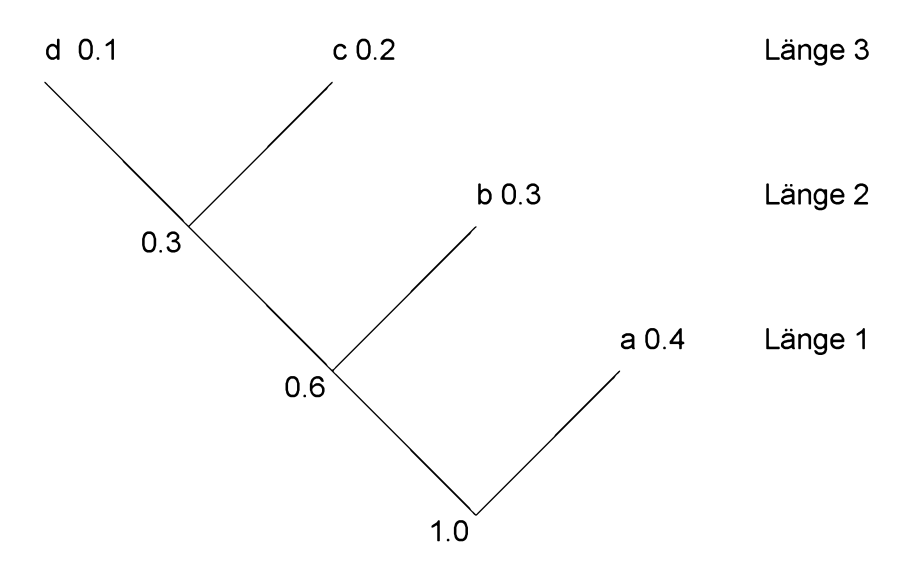
$$H = 0.4 \cdot ld(\tfrac{1}{0.4}) + 0.3 \cdot ld(\tfrac{1}{0.3}) + 0.2 \cdot ld(\tfrac{1}{0.2}) + 0.1 \cdot ld(\tfrac{1}{0.1}) \approx 1.846.$$

Codieren wir diese vier Zeichen einfach über die Sequenzen 00, 01, 10 und 11, was ein naheliegender Ansatz sein könnte, dann erhielten wir logischerweise die mittlere Länge 2:

$$L = 0.4 \cdot 2 + 0.3 \cdot 2 + 0.2 \cdot 2 + 0.1 \cdot 2 = 2.$$

Die Redundanz liegt hier also bei $R = L - H \approx 0.154$.

Bauen wir nun jedoch den Huffman-Baum nach obigem Muster auf: dann bündeln wir zuerst die Codierung für c und d zu einem Knoten mit der Gesamtwahrscheinlichkeit 0.3; anschließend wird dieser Knoten gemeinsam mit dem b zu einem weiteren Knoten zusammengefasst, die Wahrscheinlichkeit an diesem Knoten beträgt nun 0.6; schließlich ist das a noch übrig als Alternative mit der relativen Auftrittshäufigkeit 0.4. Das heißt: bis auf Spiegelung sieht der Huffman-Baum wie folgt aus.



Wird nach links die "0" und nach rechts die "1" aufgetragen, ist der Baum so zu interpretieren: das 'a' wird codiert durch "1", das 'b' durch "01", das 'c' durch "001" und das 'd' durch "000". Die mittlere Länge L berechnet sich in diesem Fall zu:

$$L = 0.1 \cdot 3 + 0.2 \cdot 3 + 0.3 \cdot 2 + 0.4 \cdot 1 = 1.9.$$

Die Redundanz liegt hier also bei $R = L - H = 0.1$. Das bedeutet: über größere Datenmengen gerechnet sparen wir mit dieser Codierung Platz!

1.4.3. Fehler bei Speicherung und Übertragung

Da auch ein DV-System nicht perfekt ist, treten mitunter Fehler bei der Speicherung oder Übertragung von Daten (Information, Nachrichten) auf[20].

Geht es um eine Ja-Nein-Frage, die entsprechend ohne Redundanz mit 0 oder 1 codiert wird, so gibt es "kein Pardon": tritt hier in nur einem Bit ein Übertragungsfehler auf, so ist die Antwort definitiv falsch - und sie kann grundsätzlich nicht als falsch erkannt werden!

[20] Aus Sicht der Codierungstheorie ist es letztlich unwichtig, ob wir über Probleme bei der Speicherung oder Übertragung sprechen, da es in beiden Fällen auf die Rekonstruktion der ursprünglichen "korrekten" Daten aus einer eventuell "korrupten", fehlerbehafteten Datenmenge ankommt.

Daher nimmt man in der Praxis Redundanz ganz bewusst in Kauf (bis zu einem gewissen Grad), denn nur so können Fehler erkannt - oder vielleicht sogar vollautomatisch korrigiert - werden.

1.4.3.1. Einige Code-Beispiele

Betrachten wir die folgenden Beispielcodierungen über dem schon bekannten übersichtlichen Alphabet $A := \{ a, b, c, d \}$.

a) C1: $c_1(a) := 00$, $c_1(b) := 01$, $c_1(c) := 10$, $c_1(d) := 11$;

b) C2: $c_2(a) := 000$, $c_2(b) := 011$, $c_2(c) := 101$, $c_2(d) := 110$;

c) C3: $c_3(a) := 0000$, $c_1(b) := 0101$, $c_1(c) := 1010$, $c_1(d) := 1111$;

d) C4: $c_1(a) := 1$, $c_1(b) := 01$, $c_1(c) := 001$, $c_1(d) := 000$.

Treten bei Code C1 (bitweise) Übertragungsfehler auf, so ist "nichts zu machen", d.h. es kann kein Fehler analysiert werden, weil jede Bitfolge eine gültige Codierung repräsentiert.

Anders bei Codierung C2. Offenbar werden hier mehr Bits als eigentlich notwendig für die vier verschiedenen Möglichkeiten eingesetzt. Im konkreten Fall ist das dritte Bit stets so gewählt, dass die Summe aller Bits (modulo 2 gerechnet) 0 ergibt; anders formuliert: die Quersumme ist gerade bzw. (modulo 2 gelesen) 0. Tritt hier ein einzelner Bitfehler auf, tritt beispielsweise die Sequenz 111 auf, so kann sofort festgestellt werden, dass diese Belegung ungültig ist. Allerdings kann sie nicht automatisch korrigiert werden, denn es könnte sein, dass bei 101 versehentlich das mittlere Bit fehlerhaft übertragen worden ist oder bei 110 das letzte.
Man spricht bei den durch die Codierung verwendeten Sequenzen, hier z.B. 011, von *Nutzwörtern*; im Gegensatz dazu heißen die nicht verwendeten Sequenzen wie in diesem Fall etwa 010 *Fehlerwörter*.

Sehen wir uns die "spendable" Codierung C3 an. Hier werden doppelt so viele Bits genutzt wie nötig wären. In diesem simplen Fall wird das 2-Bit-Muster aus Codierung C1 einfach gedoppelt. Tritt hier nun ein 1-Bit-Fehler auf, etwa 1011, so kann wiederum der Fehler als solches festgestellt werden. Interessanterweise kann aber auch hier der Fehler nicht maschinell korrigiert werden, denn die fehlerhafte Sequenz kann durch eine 1-Bit-Abweichung von 1010 oder von 1111 resultieren!

Codierung C4, die wir im vorherigen Unterkapitel im Kontext des Huffman-Baumes kennengelernt haben, bereitet uns überhaupt keine Freude, denn wird eine Sequenz der Art 1 001 01 übertragen und tritt hierbei ein Bitfehler auf - beispielsweise 1 011 01 -, so ist "fast nie" ein Fehler festzustellen, da nahezu alle Sequenzen gültig sind. Lediglich singuläre Ausreißer wie die Muster 00 oder 00000 könnten als Fehler erkannt werden, wobei wiederum fraglich bleibt, was das ursprünglich korrekte Ausgangsmuster gewesen sein soll.

Um das Ganze etwas systematischer anzugehen wieder - Sie ahnten es schon - einige formale Definitionen.

1.4.3.2. Definition (Stellendistanz, Hamming-Distanz)

a) Gegeben seien zwei gleichlange Zeichenketten s, t über demselben Alphabet A. Dann wird als *Stellendistanz* $d(s,t)$ die Anzahl der unterschiedlichen Bits auf den jeweiligen Positionen von s und t bezeichnet. Mathematisch:
$d(s,t) := \#\{\ i \mid s_i \neq t_i,\ 1 \leq i \leq n\ \},\ s = s_1 s_2 \ldots s_n,\ t = t_1 t_2 \ldots t_n$.

b) Es sei $c: A^* \to B^*$ ein homomorpher Code, der für einzelne Zeichen aus A stets gleichlange Codewörter liefert. Das heißt, diese Codierung kann auch als $c: A \to B$ aufgefasst werden, da wegen der Homomorphie hierdurch der komplette Code bereits definiert ist.
Die *Hamming-Distanz* d_H eines solchen Codes $c: A^* \to B^*$ bzw. $c: A \to B$ ist definiert als die kleinste auftretende Stellendistanz zweier Codewörter von c.
Mathematisch: $d_H(c) := \min\{\ d(c(a_1), c(a_2)) \mid a_1, a_2 \in A,\ a_1 \neq a_2\ \}$

Betrachten wir die beiden Bitmuster $s := 0011$ und $t := 0100$, dann ist die Stellendistanz $d(s,t) = 3$. Für die Codierung C2 aus dem obigen Beispiel ergibt sich die Hamming-Distanz $d_H(C2) = 2$, denn je zwei Codewörter sind um genau zwei Bits verschieden. (Bitte nachrechnen!)

Auch für die o.g. Codierung C3 ist die Hamming-Distanz $d_H(C3) = 2$, auch wenn es einzelne Bitmuster gibt, die eine Stellendistanz von 4 zueinander besitzen.

Natürlich ist die Hamming-Distanz eines jeden Codes mindestens gleich 1, da ansonsten zwei verschiedene Zeichen aus A (oder Worte über A) auf dasselbe Ergebnis abgebildet würden.

Es gilt das allgemeine Resultat: hat ein Code C eine Hamming-Distanz $d_H(C) = h$, so können Fehler (Abweichungen) von maximal $h-1$ Bits erkannt und von maximal $\frac{h-1}{2}$ Bits korrigiert werden.

Damit ist auch begründet, weshalb auch der obige Code C3 keine Fehlerkorrektur ermöglicht, denn hier ist die Hamming-Distanz 2 und nach endlich langer Rechnung sieht man: $\frac{2-1}{2} = \frac{1}{2} < 1$.

Sie werden im nächsten Kapitel den BCD- und den ASCII-Code kennenlernen. Vielleicht überlegen Sie sich in beiden Fällen, welche Hamming-Distanzen diese Codes besitzen?

1.4.3.3. Prüfziffer-Systeme

Soll ein DV-System insbesondere Fehler bei der Eingabe - etwa von numerischen Angaben wie Konto- oder Artikelnummern - erkennen und zurückmelden können, dann wird in der Regel ein Prüfziffernsystem eingesetzt. Der Code C2 aus dem vorherigen Beispiel 1.4.3.1. nutzt eine ganz einfache Prüfziffer: die sogenannte Parität.

Andere Beispiele sind etwa die zehnstelligen *ISBN* von Büchern, bei denen die ersten neun Stellen (aus dem Bereich von 0 bis 9) gemäß der nachstehenden Formel miteinander verrechnet und um eine Prüf-"Ziffer" p aus dem Bereich von 0 bis 10 ergänzt werden. Im speziellen Fall p=10 wird ein 'X' notiert. p ist so zu bestimmen, dass

$$(10a_1 + 9a_2 + 8a_3 + 7a_4 + 6a_5 + 5a_6 + 4a_7 + 3a_8 + 2a_9 + p) \bmod 11 = 0$$

gilt.

Ähnlich geht man beim einheitlichen Kontonummernsystem *EKONS* vor. Hier wird eine neunstellige Ausgangssequenz $a_1a_2...a_9$ um eine Prüfziffer p aus dem Bereich 0 bis 9 so ergänzt, dass

$$(2a_1 + a_2 + 2a_3 + a_4 + 2a_5 + a_6 + 2a_7 + a_8 + 2a_9 + p) \bmod 10 = 0$$

gilt.

1.4.3.4. m-aus-n-Codes

Als *m-aus-n-Code* (für natürliche Zahlen m und n mit $m < n$) bezeichnet man Block-Codes, bei denen jeweils genau m der n Bits auf 1 gesetzt werden.

Beispielsweise stellt die nachfolgende Zuordnungsvorschrift von $A := \{ 0, 1, ..., 9 \}$ in die Menge B der 5-Bit-Sequenzen einen 2-aus-5-Code dar.

$c(0) := 00011$, $c(1) := 00101$, $c(2) := 00110$, $c(3) := 01001$, $c(4) := 01010$,
$c(5) := 01100$, $c(6) := 10001$, $c(7) := 10010$, $c(8) := 10100$, $c(9) := 11000$.

Hierbei haben zwei Nutzwörter stets eine Distanz von mindestens 2, so dass die Hamming-Distanz eines solchen Codes 2 ist[21]. Damit können bei m-aus-n-Codes 1-Bit-Fehler stets erkannt, jedoch nicht generell korrigiert werden.

Generell gibt es gerade $\binom{n}{m}$ verschiedene nutzbare Codewörter in einem m-aus-n-Code. (Weshalb?)

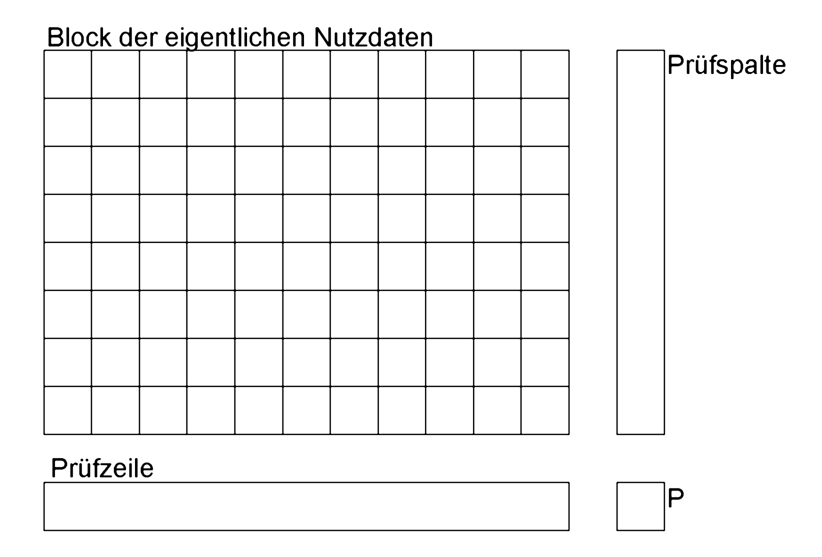
1.4.3.5. Fehlerkorrigierende Codes

Als fehlerkorrigierenden Code bezeichnet man einen Code, bei dem (mindestens) 1-Bit-Fehler nicht nur erkannt, sondern sogar vollautomatisch korrigiert werden können. Detaillierter definiert man den Begriff des n-Bit-fehlerkorrigierenden-Codes, der eine gegebene Anzahl von n Bit-Abweichungen selbständig korrigieren kann.

Als ein Beispiel eines solchen fehlerkorrigierenden Codes (für eine 1-Bit-Abweichung) sei das Prinzip der "zweidimensionalen Prüfbits" illustriert.

---

[21] Für die mathematische Präzision: Insbesondere gibt es auch zwei Nutzworte, die um eine Stellendistanz von 2 auseinander liegen.



Gemäß der obigen Skizze gibt es einen zweidimensional notierten Block von Nutzdaten (Menge einzelner Bits). Zeilenweise werden "rechts" in einer sogenannten Prüfspalte Kontrollbits errechnet und gespeichert, z.B. so, dass die Zeilensumme modulo 2 stets gerade ist. Analog wird spaltenweise verfahren und ein Kontrollbit wird in der Prüfzeile abgelegt. Schließlich gibt es ein einzelnes Prüfbit P, das einen entsprechenden Kontrollwert der Prüfzeile bzw. der Prüfspalte darstellt[22].

Tritt nun in dem gesamten Block eine Abweichung von einem Bit auf, dann kann zunächst einmal festgestellt werden, *dass* ein Fehler aufgetreten ist. Anhand der (logischen) zweidimensionalen Struktur kann jedoch sogar der Ort der Abweichung ermittelt werden.

Verschiedene Fälle sind denkbar:

a) Die Abweichung befindet sich im Nutzdatenblock. In diesem Falle weichen die Prüfbits in der betreffenden Zeile und der betreffenden Spalte ab, dadurch sind die Koordinaten bekannt und das betreffende fehlerhafte Bit kann automatisch invertiert werden.

b) Befindet sich die Abweichung in der Prüfzeile, so wird in der Prüfzeile eine Paritätsabweichung festgestellt, jedoch keine in der Prüfspalte. Nun kann durch spaltenweise Kontrolle festgestellt werden, welche Spalte fehlerhaft ist: damit ist die fehlerhafte Stelle in der Prüfzeile gefunden, diese kann wiederum selbständig korrigiert werden.
   Analog dazu ist der Fall einer Abweichung in der Prüfspalte.

c) Schließlich könnte das "Gesamtprüfbit" P fehlerhaft sein. Da die Paritätschecks der Prüfzeile und der Prüfspalte jedoch korrekt sind, wird in diesem Falle P als fehlerhaft erkannt, da es weder die Parität über die Prüfzeile, noch die über die Prüfspalte erfüllt.

---

[22] Man sieht leicht, dass bei einer korrekten Datenspeicherung das Prüfbit über die Prüfzeile stets dasselbe sein muss wie das über die Prüfspalte!

1.4.3.6. Fehlertolerante Codes

Abschließend sei kurz auf die sogenannten *fehlertoleranten Codes* hingewiesen. Darunter versteht man Codes, bei denen Abweichungen in ein oder wenigen Bits zwar nicht erkannt werden (müssen), bei denen eine solche Abweichung jedoch in *fachlicher* Hinsicht nicht gravierend ist.

Insbesondere im Kontext von Zahlencodes, also der Codierung von Zahlwerten, geht man diesen Weg. Zahlencodes, bei denen "benachbarte" Bit-Sequenzen auch benachbarte Zahlen darstellen, nennt man *Gray-Codes*.

Ein Beispiel für einen solchen Gray-Code für die Ziffern 0 bis 9 wird nachstehend aufgeführt.

dezimal	0	1	2	3	4	5	6	7	8	9
Gray-Code	0000	0001	0011	0010	0110	1110	1111	1101	1100	1000
binär	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Wer mehr hierzu lesen möchte, sei auf das Buch von [Ernst], S. 77ff, verwiesen.

1.5 Praktische Codierung: Zahlen, Zeichen, Dateien und Grafiken

Nach den Einblicken in die Codierungstheorie wollen wir uns nachstehend mit einigen praktischen Aspekten der Codierung befassen; exemplarisch behandeln wir Möglichkeiten der Zahlencodierung, der Codierung von einzelnen Zeichen, ganzen Dateien und - gewissermaßen als Spezialfall - von Grafiken.

1.5.1 Codierung von Zahlen

Neben der eingangs diskutierten Codierung von ganzen (positiven) Zahlen gibt es noch eine ganze Reihe anderer „Dinge", die gespeichert, also auch codiert werden müssen.

Beginnen wir mit der Abspeicherung verschiedener Arten von Zahlen in sogenannten Zahlformaten.

Das Prinzip (bzw. die Codierungsvorschrift oder das *Format*), eine übliche, nichtnegative ganze Zahl (wie etwa 12) einfach durch ihre Dualentwicklung zu repräsentieren, nennt man „echte Dualzahl".

So ist die Dezimalzahl 12 (eindeutig) zerlegbar in die Dualentwicklung $12_{dez} = 8 + 4 = 1 \times 2^3 + 1 \times 2^2 + 0\ 2^1 + 0 \times 2^0 = 1100_{dual}$.

Das Format der echten Dualzahl eignet sich typischerweise gut für alle Arten von „Rechenaufgaben", in diesem Format rechnet der Computer selbst.

Steht mehr die Ausgabeformatierung im Vordergrund für Darstellungen auf einem Bildschirm oder Ausdruck, dann kann eine anderes Format gewählt werden: das sogenannte „unechte Dualzahlformat". Hierbei wird im Falle der 12 nicht der numerische Wert 12 dual

ermittelt und abgespeichert, vielmehr wird die Darstellung als Zeichenfolge '1' '2' auch intern abgespeichert.

Die Zahl 12 wäre demnach als unechte Dualzahl abgespeichert die Folge der zwei ASCII-Zeichen[23] '1' und '2'; ein Blick in eine ASCII-Tabelle[24] verrät, dass die '1' den ASCII-Code Nr. 49, die '2' die 50 besitzt. Wird dies nun noch in das Dualsystem umgewandelt, so entsteht die Darstellung der 12 als unechte Dualzahl[25]: 0011 0001 0011 0010.

Sieht man sich die (auf ASCII basierenden) unechten Dualzahlen genauer an, dann stellt man fest, dass das obere Halbbyte stets 0011 lautet; dies könnte somit auch weggelassen werden, sofern der verarbeitenden Software „klar" ist, dass es sich bei den betreffenden Daten um das jeweils zweite Halbbyte einer solchen unechten Dualzahl handelt.

Auf diese Weise gelangt man zur sogenannten „gepackten unechten Dualzahl": die Zahl 12 wäre somit zu speichern als nur noch zwei Halbbytes große Sequenz 0001 0010.

Zu diesem Verfahren gelangt man auch, wenn man direkt dem Ansatz der sogenannten „*BCD-Zahlen*" (*binary coded decimals*) folgt: Für das Abspeichern einer Ziffer im Zehnersystem sind offensichtlich vier Bits notwendig, denn mit drei Bits kommt man nur gerade auf $2^3 = 8$ Möglichkeiten. Somit ist ein naheliegender Ansatz, jeweils eine Ziffer des Dezimalsystems in einer sogenannten *Tetrade*, also einer Sequenz von vier Bits abzuspeichern. Damit sind die Sequenzen 0000, 0001, 0010, usw. bis 1001 „verbraucht" für die Ziffern '0' bis '9'. Die restlichen sechs Möglichkeiten (1010 bis 1111) werden nicht benötigt, hier spricht man daher von *Pseudotetraden*.

Ein solcher *BCD-Code* (*binary coded decimal*) arbeitet nach diesem Tetraden-Prinzip[26]. Die mehrstellige Dezimalzahl 234 wäre somit darstellbar durch die Tetradensequenz 0010 0011 0100. Und in unserer vorherigen Sprechweise wäre dies eine gepackte unechte Dualzahl.

Geht man weiter, so sind als nächstes auch negative ganze Zahlen zu speichern. Dies geschieht dadurch, dass das höchste Bit, also das skizziert am weitesten links stehende, als sogenanntes Vorzeichenbit verwendet wird; es ist 0 für Zahlen größer oder gleich 0 und 1 für negative Zahlen.

Die bis hier diskutierten Zahldarstellungen kann man unter dem Begriff der Festkommazahl oder Festpunktdarstellung zusammenfassen, wenn man sich folgende naheliegende Erweiterung vor Augen führt.

Bislang haben wir stets ganze Zahlen (wie etwa 12) dargestellt; mit derselben Technik könnte man aber ohne weiteres auch z.B. Zahlen mit einer festgelegten Anzahl Nachkommastellen codieren, wenn nur festgelegt ist, wo das Komma stehen soll. Einigt man sich beispielsweise auf zwei Nachkommastellen (wie bei Preisangaben üblich), dann wäre die Zahl 12 (Mark oder Euro) als 1200 (Pfennig oder Cents etc.) zu speichern. Die Abspeicherung

---

[23] Hier verwenden wir bereits den Begriffs des ASCII-Codes, der erst auf Seite 36 ausführlicher behandelt werden wird. Im Moment ist lediglich notwendig zu wissen, dass jedes Zifferszeichen (wie etwa die '1') durch einen eindeutigen (und zunächst willkürlichen) Code repräsentiert wird.

[24] Eine solche findet sich zufällig im Anhang dieses Skripts auf Seite 237.

[25] Ggf. ist nachzurechnen: 49_{dez} = $0011\ 0001_{dual}$, 50_{dez} =$0011\ 0010_{dual}$!

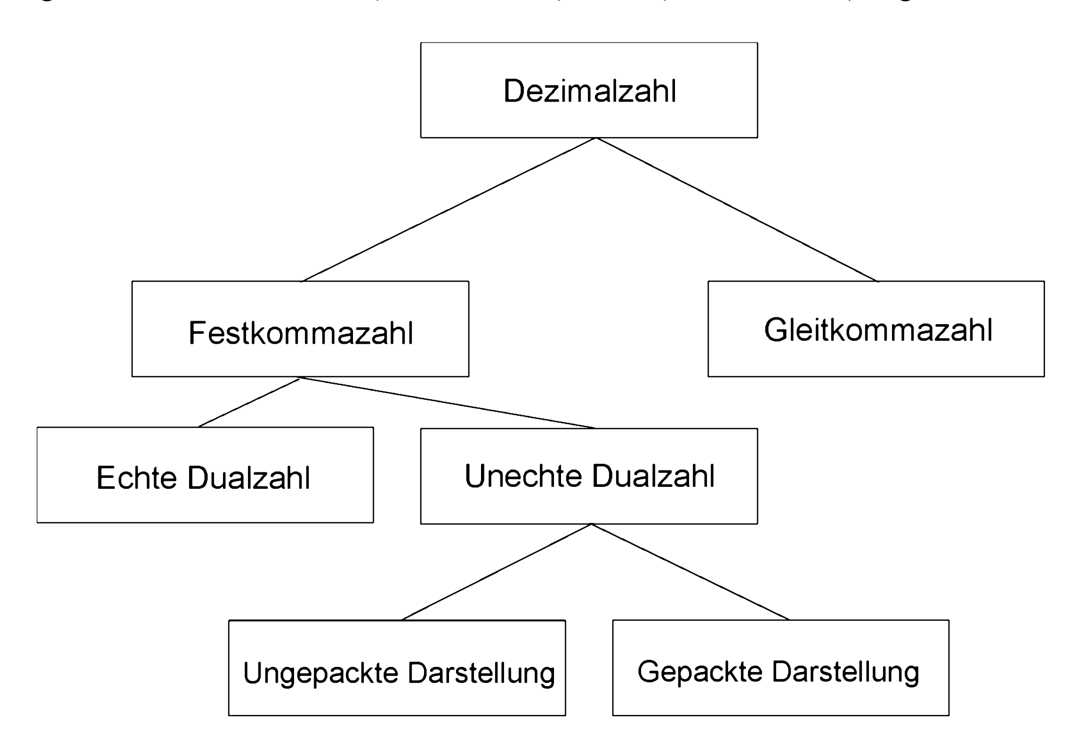
[26] Neben dem BCD-Code gibt es noch einige andere Tetradencodes, z.B. der Aiken- oder der Gray-Code. Auf diese soll hier jedoch nicht näher eingegangen werden; stattdessen sei auf [Dworatschek] S. 169 ff verwiesen.

0001 0010 0011 entspräche somit der Sequenz 1 2 3 (dezimal gelesen), und zusammen mit der Festlegung des Dezimalkommas stünde dies für 1,23 (eins komma zwei drei).

Bei diesem Codierungsschema steht die Anzahl der Nachkommastellen und die der Vorkommastellen von Anfang an fest; in unserem Beispiel können Zahlen nur bis zur zweiten Nachkommastelle „genau" gespeichert werden. Eine Zahl wie etwa 1,2345 müsste zwangsläufig gerundet (als 1,23) abgespeichert werden.

Nun gibt es aber auch Problemstellungen, bei denen nicht von vorne herein klar ist, in welchen Größenordnungen und mit welcher Genauigkeit eine Zahl darzustellen ist. So ist es gelegentlich erforderlich, auch die Zahlen 1.230.000 oder 0,0000000000000123 abzuspeichern.

Hierzu dient die Gleitpunkt- oder Gleitkommadarstellung. Eine Zahl x wird mathematisch dargestellt als $x = m \times B^e$, wobei m als die *Mantisse*, B die *Basis* und e der *Exponent* (oder die sogenannte *Charakteristik*) bezeichnet wird. In der Regel wird als Basis B der Wert 2 gewählt. Eine Zahl x ist somit zerleg- und darstellbar als $x = m \times 2^e$. Für die Mantisse wird festgelegt, dass diese zwischen 0 (einschließlich) und 1 (ausschließlich) liegen muss.



Damit wäre die Zahl $x = 12_{dez} = 1100_{dual} = 0{,}1100_{dual} \times 2^4$ darstellbar mit Vorzeichen 0 (Zahl nicht-negativ), Mantisse 1100_{dual} und Exponent $100_{dual} = 4_{dez}$.

1.5.2. Codierung von Zeichen

Während es bei einer ganzen Zahl wie 0, 1 oder 2 nach dem bisher Gesagten nun klar sein sollte, wie man diese auf einem binär orientierten (digitalen) DV-System abspeichert, ist dies bei einem Zeichen wie 'A' keineswegs geklärt. Da der Digitalrechner, salopp formuliert, nur 0 und 1 als atomare Bausteine speichern kann, kann er intern „in Wahrheit" nur endliche Sequenzen von 0 und 1 darstellen.

Daraus resultiert die Notwendigkeit der Codierung. Unter einem *Code* versteht man in diesem Zusammenhang die Abbildung des darzustellenden Zeichenvorrates auf eine entsprechende Menge endlicher 0-1-Folgen.

Würde man nur die vier Zeichen 'A', 'B', 'C' und 'D' speichern wollen, so könnte man einen 2-Bit-Mini-Code vereinbaren, bei dem 'A' durch 00, 'B' durch 01, 'C' durch 10 und 'D' durch 11 dargestellt werden. Der bekannte Automobilclub „ADAC" wäre dann sehr einfach darzustellen als Sequenz 00 11 00 10. Die beliebte Rockgruppe „AC/DC" wäre (ohne den Schrägstrich allerdings) darzustellen in der Form 00 10 11 10.

Die hier der besseren Lesbarkeit eingeschobenen Leerzeichen muss bzw. kann der Rechner natürlich nicht mitspeichern, dies wäre ja ein weiteres, fünftes Zeichen, das in 2 Bit nicht mehr hineinpasst. Solange aber die Breite (=Anzahl der Bits) eines Zeichens in einem bestimmten Code gleich ist[27], benötigt man natürlich auch keine Trennmarkierungen zwischen den einzelnen Zeichen.

In der Praxis benötigt man natürlich mehr Zeichen, und es wurden in der Vergangenheit auch verschiedene Codierungen festgelegt.

Es sei an dieser Stelle kurz erwähnt, dass es noch zahlreiche weitere Codierungsformen gibt, u.a. auch fehlererkennende und fehlerkorrigierende Codes, bei denen naturgemäß mit Redundanz gearbeitet wird.

Ein gängiges Verfahren einer fehlererkennenden Codierung ist die Verwendung eines Prüfbits. Sollen beispielsweise drei Bits (= acht Möglichkeiten) übertragen werden, so hängt man ein viertes Bit an, das z.B. so gewählt wird, dass die Quersumme immer gerade ist (*even parity check*). Zur 3-Bit-Folge 101 würde also das Prüfbit 0 treten: 1010 hat eine gerade Quersumme. Die 3-Bit-Folge 010 würde mit 1 ergänzt zu 0101 - wiederum einer Sequenz mit gerader Quersumme.

Würde nun bei z. B. einer Datenübertragung die Sequenz 1110 auftreten, so wäre im hier zugrundegelegten Szenario klar, dass ein Übertragungsfehler stattgefunden haben muss, denn die Quersumme ist nicht geradzahlig. Hiermit kann nun keine Fehlerkorrektur stattfinden, aber der Empfänger der Daten kann prinzipiell den Sender auffordern, die (korrekte) Sequenz noch einmal zu schicken.

Die für den PC-Bereich relevante Zeichen-Codierung geschieht mittels des *ASCII*, des *American Standard Codes for Information Interchange*. Hierbei handelt es sich um einen laut Standard sieben Bit breiten Code, der die (US-amerikanischen) Zeichen beinhaltet. Darüber hinaus wird als erweiterter 8-Bit-ASCII der um jeweils nationale oder sonstige Sonderzeichen erweiterte Code verstanden.

Unter Microsoft Windows kann der Code sehr schön in dem Zusatzprogramm *(Unicode-)Zeichentabelle* betrachtet und eingesetzt werden. Im nachstehenden Bild wird der 8-Bit-ASCII für Deutschland (in der Schriftart Times New Roman) dargestellt. Mit der Maus wurde das Zeichen mit der Code-Nummer 159, das Ÿ, markiert.

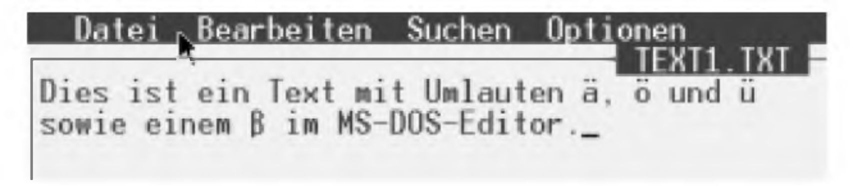
---

[27] Dies muss nicht zwingend immer so sein. Kompressionsverfahren oder auch das bekannte Morse-Alphabet arbeiten gerade damit, dass häufig vorkommende Zeichen mit einer geringeren Bitbreite dargestellt werden als seltener auftretende.
So ist im Morse-Alphabet der häufig auftretende Buchstabe 'e' codiert mit . (einem Punkt), das sehr viel seltener auftretende 'x' hat den Morse-Code - . . - (Strich Punkt Punkt Strich).



Zu beachten ist, dass der ASCII-Code nur auf 7 Bit standardisiert ist. Das heißt: ein Text mit deutschen Umlauten auf einem PC kann bzw. wird auf einem Apple Macintosh nur noch verstümmelt erscheinen: die Umlaute und ‘ß’ werden als irgendwelche anderen Sonderzeichen auf dem Macintosh wiedergegeben werden. Ähnlich verhält es sich mit Daten, die von einer UNIX-Anlage auf einen PC überspielt werden. Aber auch DOS und Windows benutzen zwei verschiedene ASCII-Varianten, die von Windows wird mit ANSI-Code bezeichnet.

Die nachstehenden beiden Bildschirmabzüge zeigen einen kleinen Text, einmal im MS-DOS-Editor (DOS-ASCII) und einmal, ohne dass eine Änderung an der Datei stattgefunden hätte, in der Windows-Textverarbeitung Lotus Word Pro (ANSI-Code).





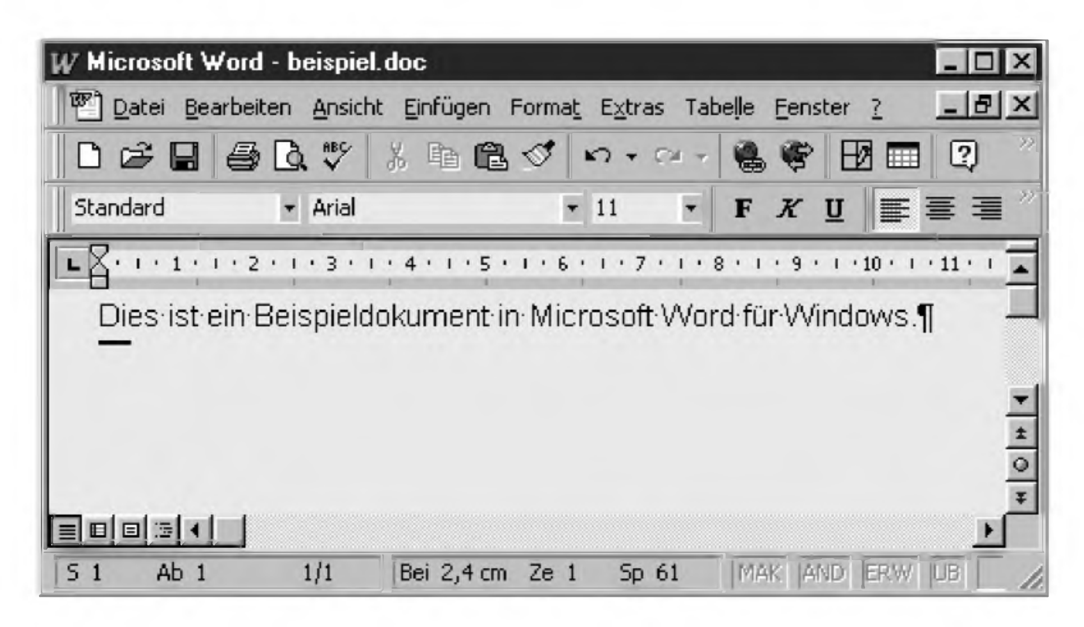
Neben dem ASCII-Code ist noch der *EBCDIC* (*extended binary coded decimal information code*) zu erwähnen, der in der IBM-Großrechnerwelt verwendet wird.

1.5.3. Codierung von Dateien

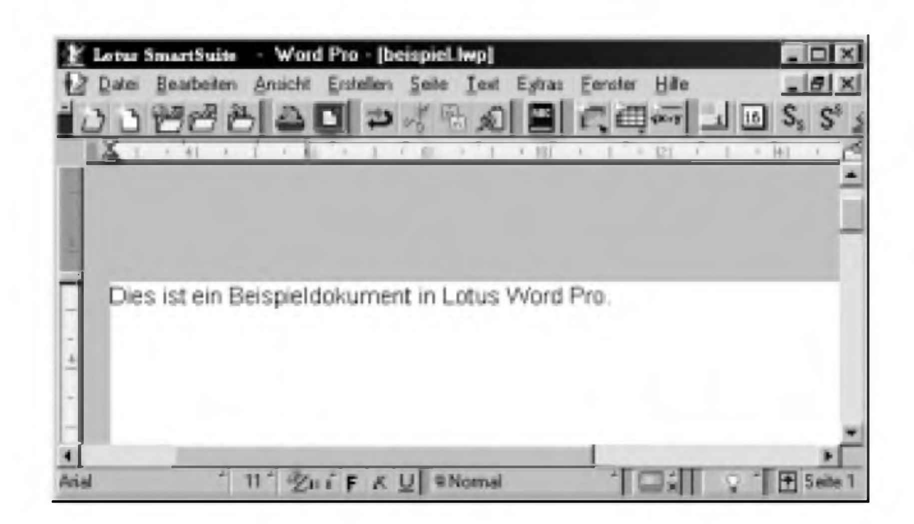
Sehr viel komplexer ist die Codierung, wenn man sich nicht um einzelne Zeichen oder Zahlen kümmert, sondern ganze Dateien, inbesondere Graphiken, behandeln möchte. Hier gibt es eine Vielzahl sogenannter Datei- und Graphikformate; einige werden, um einen gewissen Überblick herzustellen, im Anhang A.2. auf Seite 238 tabellarisch aufgelistet.

Im wesentlichen entscheidet sich jeder Software-Hersteller zunächst einmal für ein eigenes, *proprietäres* Format. Um seine Kunden bei der Stange zu halten, wird nichts unternommen, was einen Wechsel zu einer anderen Software erleichtern würde. Wären die Dateiformate der einzelnen Programme offengelegt, so könnte beispielsweise jeder Anwender der Textverarbeitungssoftware Microsoft Word auf StarWriter wechseln (oder umgekehrt) und könnte dabei seine alten Dateien unverändert öffnen und weiterbearbeiten.

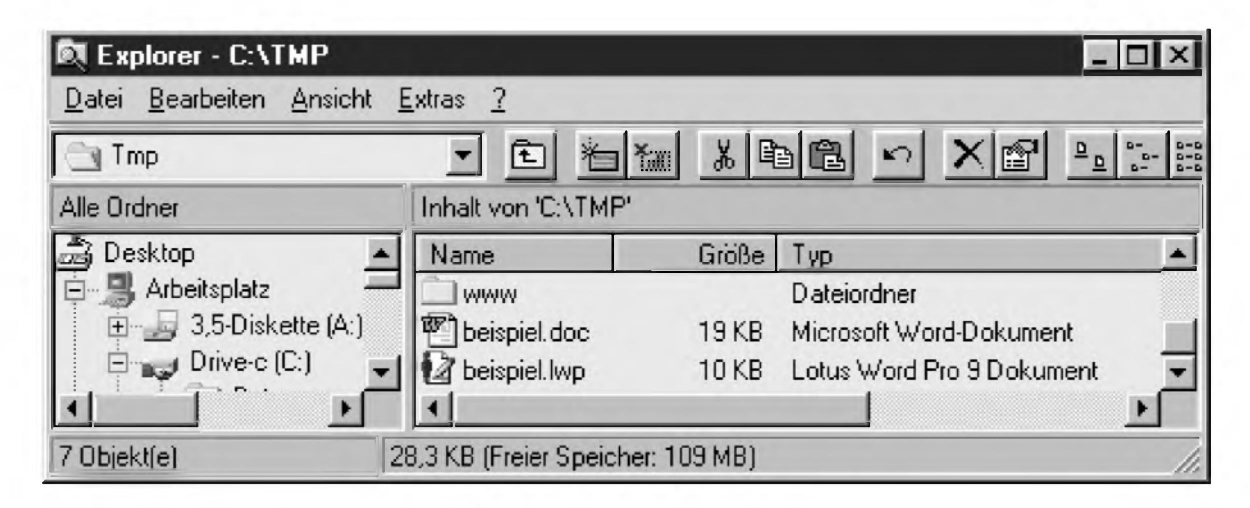
Sehen wir uns jeweils ein Dokument in den beiden Textverarbeitungsprogrammen Microsoft Word für Windows (beispiel.doc) und Lotus Word Pro (beispiel.lwp) an.



Beide Dokumente enthalten (als bewußte Benutzereingabe) nur jeweils einen Satz, wie auf den Bildschirmabzügen gezeigt. Doch schon die Dateigröße der beiden Dokumente differiert erheblich.



Während in dieser konkreten Situation die Word-Datei 19 Kilobyte groß geworden ist, beansprucht die Lotus Word Pro-Datei „nur" 10 Kilobyte. Gleichzeitig ist allerdings auch zu bemerken, dass der eigentlich gespeicherte Text gerade mal um die 50 Bytes groß ist!



Das heißt: in beiden Fällen, Word wie Word Pro, wurde über den eigentlichen Text hinaus eine ganze Menge weiterer Information gespeichert. Und das ist auch nachvollziehbar, denn in einer Textverarbeitung werden u.a. auch Attribute wie die Schriftart, die Schriftgröße, ggf. die Farbe des Textes, Eigenschaften der gesamten (Papier-)Seite usw. mitverwaltet und dementsprechend auch gespeichert.

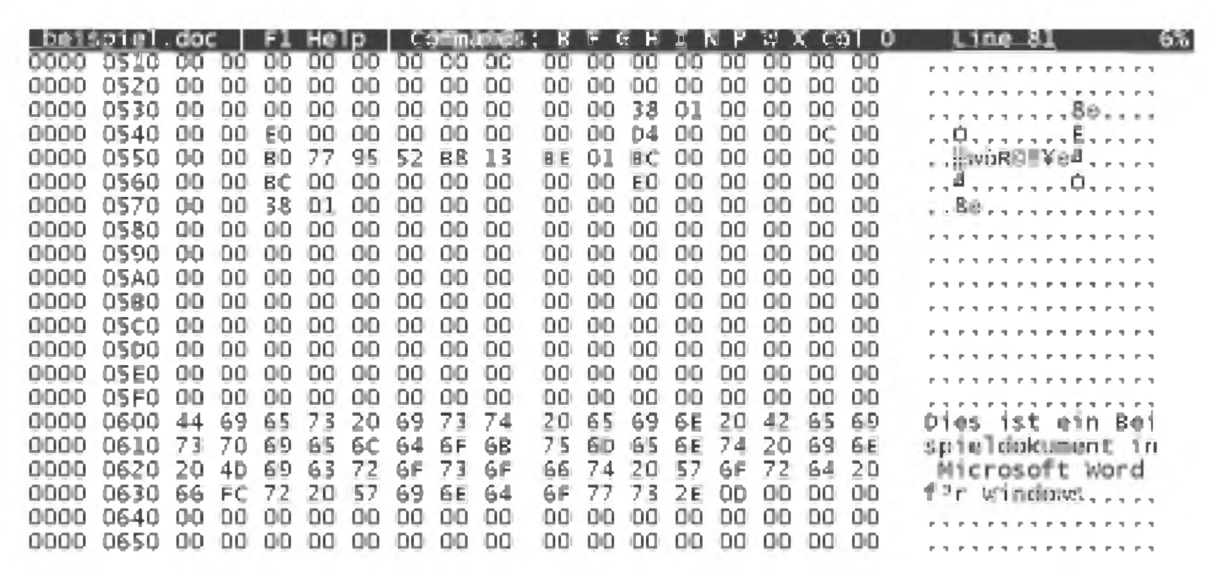
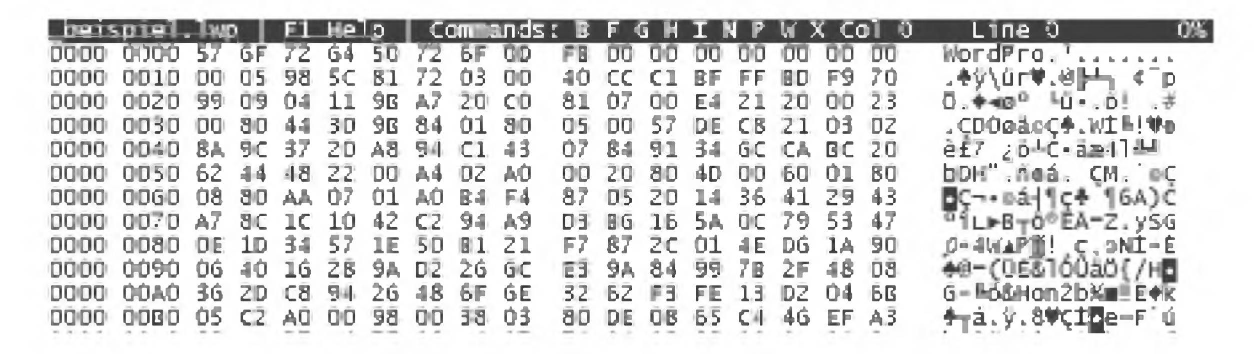


Bild: Innenansicht einer Microsoft Word-Datei[28]

Sehen wir einmal spaßeshalber in das „Innenleben" der beiden Dateien hinein: im vorherigen Bild sehen wir „in die" Word-Datei, der nachfolgende Schnappschuss zeigt eine Innenansicht der Datei von Lotus Word Pro. In beiden Fällen sehen wir, dass eine ganze Menge uns (als „normalen Menschen") unverständlicher Zeichen gespeichert ist; bei dem Ausschnitt der Word-Datei sehen wir offensichtlich auch die eigentliche Textpassage, bei dem Ausriss der Word Pro Datei können wir offensichtlich nur erkennen, dass es sich um eine WordPro-Datei handelt, denn diese Information ist zu Beginn der Datei in Klarschrift gespeichert.



Bereits seit langem arbeiten immer wieder Standardisierungskomitees daran, einheitliche und vor allem offengelegte Dateiformate zu definieren und zu verbreiten. Teilweise beteiligen sich auch namhafte Unternehmen dabei. Erwähnt werden sollen hier drei offene Text- bzw. Dokumentformate.

Rich Text Format

Zum einen gibt es von der Firma Microsoft das Rich Text Format (RTF). Dieses wurde explizit für den offenen Datenaustausch mit Textverarbeitungssoftware anderer Hersteller, aber auch zum Datenaustausch zwischen verschiedenen Plattformen, erfunden.

---

[28] Für Interessierte: Dieses und das folgende Bild zeigen den Inhalt einer Datei, wobei in den ersten beiden Spalten hexadezimal eine Speicheradresse angegeben ist, danach folgen zeilenweise jeweils zweimal acht Byte, dargestellt durch den hexadezimalen Wert; in der Spalte ganz rechts schließlich ist zu sehen, was als ASCII-Zeichen von all diesen Zeichen darstellbar ist. Nicht darstellbare Zeichen sind mit einem Punkt aufgeführt.

Das heißt: wenn Sie einen Text schreiben, der im wesentlichen über die gängigen Atttribute wie „fett”, „kursiv”, „unterstrichen”, Schriftarten und -größen usw. verfügt, dann können Sie diesen auch im Format RTF speichern und z.B. per eMail an Menschen versenden, die eventuell dann doch mal kein Microsoft Word für Windows (oder nicht die passende Version) zur Verfügung haben sollten.

Hypertext Markup Language

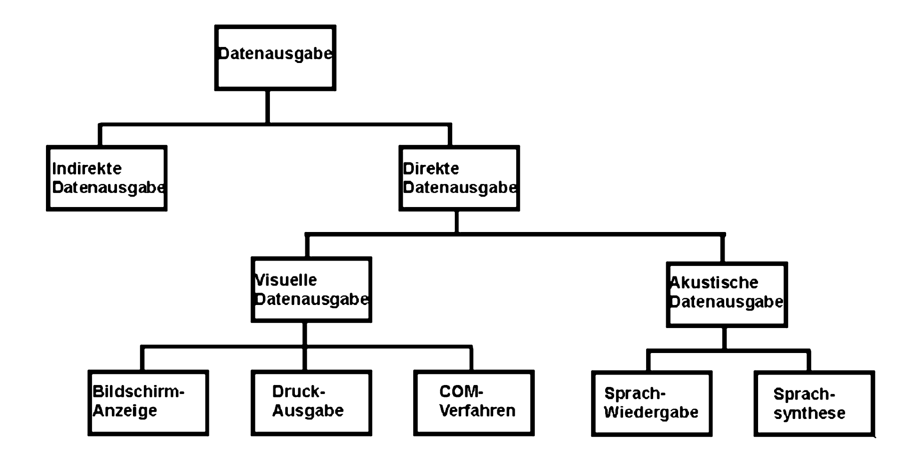
Ein weiteres, von der Natur der Sache her offenes Format ist die Seitenbeschreibungssprache des World Wide Web, HTML, die Hypertext Markup Language. Alle Internet-Browser „sprechen” diese Sprache, und wirft man einen Blick „hinein”, dann sieht man auch ohne HTML-Kenntnisse, dass dies ein offenes Format ist: man kann es mit relativ wenig Aufwand lernen. (Siehe hierzu auch Abschnitt 8.5. auf Seite 236.)

Heutzutage können fast alle modernen Textverarbeitungen Dokumente auch im HTML-Format lesen und speichern; insofern kann auch dieses Format in gewissen Grenzen als Austauschformat für (reine) Textdokumente dienen.

Acrobat Reader

Einen ganz anderen Weg hat die Firma Adobe[29] beschritten. Ausgehend von der lange währenden Problematik, dass es keinen anerkannten und offenen Standard für hochwertige DTP[30]-Dokumente gegeben hat, hat Adobe mit Acrobat einen neuen Quasi-Standard kreiert, der auch qualitativ sehr gut zur Speicherung von Dokumenten mit graphischen Elementen geeignet ist und den Ansprüchen des DTP genügt. Darüber hinaus hat Adobe durch einen für fast alle Betriebssysteme frei erhältlichen Browser, den sogenannten Acrobat Reader, dafür gesorgt, dass jedermann die (in der Regel mit der Dateiendung .pdf versehenen) Acrobat-Dokumente auch lesen (und ggf. ausdrucken) kann.

Das Acrobat-Format wird u.a. von Hochschulen für wissenschaftliche Zwecke (Skripten, Arbeiten usw.) und teilweise auch im Internet eingesetzt; daneben findet es sich häufig auf CD-ROMs für sogenannte README-Dateien und digitale Handbücher.



[29] Bekannteste Produkte von Adobe sind sicherlich die Graphikprogramme *PhotoShop* oder *Illustrator*.

[30] DTP = Desktop Publishing

1.5.4. Codierung von Graphiken

Eine spezielle und besonders vielseitige Form der Codierung stellt das Abspeichern von Graphiken (Zeichnungen, eingescannte Photos usw.) dar. Darum wollen wir uns an dieser Stelle etwas Raum nehmen, diesen Themenkomplex anzusprechen[31]. Zum Thema Graphikformate sei generell auf das Buch von [Born] verwiesen.

Man unterscheidet verschiedene Formatfamilien anhand der Art und Weise, wie die betreffende Graphik abgespeichert wird.

1.5.4.1. Rasterformate (Pixelgraphiken, Bitmaps)

Die sogenannten Rasterformate enthalten für jeden darzustellenden Punkt die Daten eines Bildes; jedem Bildpunkt oder Pixel sind seine Koordinaten und ein Farbwert[32] zugeordnet. Die Vorteile dieser Methode: Bitmap-Graphiken sind sehr einfach zu erstellen, die Pixel können von entsprechender Software leicht einzeln oder gruppenweise manipuliert werden. Außerdem „passt" dieses Format auf alle punktweise operierenden Ausgabegeräte, z. B. Drucker oder Bildschirme. Nachteilig ist anzumerken, dass Bitmapdateien sehr groß werden können, vor allem wenn sehr viele Farben und/oder eine hohe Auflösung verwendet werden. Durch Kompressionsverfahren kann dem in gewissem Maße jedoch entgegengewirkt werden.

Ein gravierender Nachteil ist allerdings, dass sich Rasterformate nur schlecht skalieren lassen; bei der Vergrößerung und der Verkleinerung besitzt das neu zu errechnende Bild durch Rundungsfehler in der Regel eine erheblich schlechtere Qualität als das Original!

1.5.4.2. Vektorformate

Bei den Vektorformaten werden graphische Objekte durch eine formale Beschreibung abgespeichert. Eine (farbige) Linie zwischen zwei Punkten kann etwa dargestellt werden als Anweisung „Linie(Startpunkt,Endpunkt,Farbwert)"; ein parallel zum Koordinatensystem liegendes (ebenfalls farbiges) Rechteck wäre beispielsweise notierbar in der Form „Rechteck(Linke-obere- Ecke,rechte-untere-Ecke,Farbwert)".

Diese Darstellungsform ist leicht skalierbar, die dargestellten Einheiten (Objekte) sind relativ leicht manipulierbar. Wird ein Kreis mit dem Durchmesser 10 cm in einem Vektorformat gespeichert, so ist auch ein Ausdruck mit dem Durchmesser 18 cm kein Problem, denn anhand der Abspeicherung „Kreis(Mittelpunkt,Durchmesser,Farbwert)" wird für den Ausdruck das so entstehende Bild neu berechnet.

Vektorformate eignen sich allerdings von Natur aus nicht für komplexe Bilder (wie etwa Photographien), die pixelweise andere Farben besitzen[33].

---

[31] Einige weitergehende Informationen zum Stichwort „Graphikformate" finden sich u.a. auf der WWW-Seite http://www.lrz-muenchen.de/services/software/grafik/grafikformate/.

[32] Im Falle einer einfachen Schwarz-Weiß-Zeichnung besteht der Farbwert lediglich aus 0 für „schwarz" oder 1 für „weiß", benötigt also nur 1 Bit.

[33] Neben Raster- und Vektorformaten sind auch sogenannte Metaformate gebräuchlich; hier werden sowohl Pixel- als auch Vektorinformationen abgespeichert. Die gängigsten Formate sind CGM (Computer Graphics Metafile), PICT (Macintosh Picture) und WMF (Windows Metafile).

1.5.4.3. Einige gebräuchliche Graphikformate

Nachstehend seien ein paar gebräuchliche Graphikformate[34] aufgeführt. Unter Extension sind die Dateinamenendungen aufgeführt, die üblicherweise unter DOS und Windows verwendet werden. Der Aspekt der Datenkompression wird hier bereits erwähnt, aber erst im folgenden Kapitel 1.6. ausführlicher behandelt.

Format	Extension	Farben, Bildgröße, Kompression usw.	Raster Vektor
Windows-Bitmap	bmp	Farbtiefe: 1-Bit (schwarz-weiß), 4-Bit (16 Farben), 8-Bit (256) oder 24-Bit (16,7 Mio.); meist keine Kompression (oder RLE); maximale Bildgröße 65536 x 65536 Pixel	R
CompuServe Graphics Interchange Format (GIF)	gif	Farbtiefe: 1 bis 8 Bit (maximal 256 Farben); Kompression: LZW; maximale Bildgröße 65536 x 65536 Pixel; Besonderheit: die modernere „89a"-Version des GIF-Formates kann auch transparente Graphiken verwalten sowie „animierte GIFs", also Dateien, in denen mehrere Bilder gespeichert sind, die von geeigneter Software als kleine „Diashow" abgespielt werden. GIF ist eines der im Internet (WWW) gebräuchlichen Formate.	R
JPEG File Interchange Format (JFIF)	jpg jpeg	Farbtiefe bis 24-Bit; Kompression: JPEG; maximale Bildgröße 65536 x 65536 Pixel; JPEG ist eines der im Internet (WWW) gebräuchlichen Formate.	R
Kodak Photo CD	pcd	Farbtiefe bis 24-Bit; kodak-eigenes Kompressionsverfahren; maximale Bildgröße 3072 x 2048 Pixel; verwendet für Photo-CD-ROMs	R
Tag Image File Format (TIFF)	tif	Farbtiefe: 1- bis 24-Bit; Kompression: keine, RLE, LZW, CCITT sowie JPEG maximale Bildgröße: ca. 4 Milliarden Bildzeilen; Besonderes: mehrere Bilder in einer einzigen Datei möglich! Dieses Format wird häufig von Scanner-Software zur Abspeicherung von Photos verwendet.	R
Postscript (PS), Encapsulated Postscript (EPS)	ps eps	PostScript ist eine Seitenbeschreibungssprache von der Firma Adobe; speziell geeignet für die Ausgabe auf dafür zugeschnittene Drucker (Post-Script-Drucker)	V
Corel Draw	cdr	Eigenes (proprietäres) Graphikformat der Firma Corel Corp. für ihr Programm „Corel Draw"	V

Ein Format zur Speicherung und Übertragung von Bildinformationen muss in erster Linie möglichst kompakt sein. Im World Wide Web[35] wurden und werden hierzu die Formate GIF

[34] Eine recht lange Liste von Graphikformaten finden Sie im World Wide Web beispielsweise auf der Seite http://web.urz.uni-heidelberg.de/doc/Unterstuetzung/Hinweise/Einzel/Grafik/Formate/index.html.

[35] Vgl. hierzu auch Abschnitt 8.4.2. (S. 236).

und JPEG eingesetzt. Während JPEG vor allem zur Abspeicherung von Photos geeignet ist, sind GIF-Dateien gut bei kleineren Zeichnungen mit wenigen Farben einsetzbar.

Seit 1995 wird mit dem „Portable Network Graphics" Format (PNG) der Versuch unternommen, Vorteile von GIF und JPEG unter einen Hut zu bringen und gleichzeitig die mit dem GIF-Format verbundenen Copyright-Probleme zu bewältigen[36]. PNG ist ein urheberrechtlich freies Format, das neben dem 256farbigen Palettenmodell verschiedene Graustufen- und Echtfarbformate mit 5 bis 16 Bit Auflösung pro Farbkomponente beherrscht. Auch eine optionale Transparenzkomponente (Alpha-Kanal) ist vorgesehen. Die Möglichkeiten des GIF-Formates, mehrere Bilder oder Animations-Sequenzen speichern zu können, sind in der Version 1.0 von PNG allerdings noch nicht realisiert worden.

Sehen wir uns abschließend ein paar Graphikdateien in den Formaten GIF und JPEG einmal praktisch an.



Zunächst soll es um eine einfache, nebenstehend abgebildete Strichzeichnung gehen: das farbige Original nutzt eine 4-Bit-Farbpalette, also einen Grundvorrat von 16 Farben. Diese Graphik wird nun gespeichert in den Formaten GIF, JPEG, PCX und PNG. Dabei treten die folgenden Dateigrößen dieser 70 x 52 Pixel großen Zeichnung auf.

```
Bytes  Dateiname
  357  HALLO.GIF
1.736  HALLO.JPG
1.493  HALLO.PCX
  319  HALLO.PNG
```

Man sieht, dass hier das moderne PNG-Format am speicherplatzsparendsten ist, dagegen kann das JPEG-Format seine Vorzüge in diesem Fall überhaupt nicht ausspielen.

Wird diese Grafik nur als Schwarz-Weiß-Bild gespeichert, ergibt sich folgende Situation.

```
Bytes  Dateiname
  249  HALLO-SW.GIF
1.325  HALLO-SW.JPG
  404  HALLO-SW.PCX
  187  HALLO-SW.PNG
```

Spielt man dieses Szenario dagegen mit einem eingescannten Photo durch, so sieht die Sache erwartungsgemäß grundlegend anders aus. Ein eingescanntes Bild wurde in den jeweiligen Formaten abgespeichert, bei JPEG lag die eingestellte Kompressionsrate bei 50%.

```
  Bytes  Dateiname
688.877  PHOTO.GIF
108.194  PHOTO.JPG
649.307  PHOTO.PCX
478.183  PHOTO.PNG
596.046  PHOTO.TIF
```

Deutlich ist zu sehen, welchen Vorteil das JPEG-Format bei Photos ausspielen kann; allerdings sei noch einmal betont, dass die Datei PHOTO.JPG verlustbehaftet gespeichert ist, d.h. in ihr stecken auch nicht mehr dieselben Informationen wie beispielsweise in der TIFF-Datei.

---

[36] Zum Thema PNG sei auf den Artikel „Bits im Bilde in der Zeitschrift „iX" verwiesen, der unter der Adresse http://www.heise.de/ix/artikel/9609116/ im WWW nachgelesen werden kann.

Interessant vielleicht noch zu erwähnen, dass es bei JPEG im Rahmen der wählbaren Kompressionsstufen zu weiteren erheblichen Platzeinsparungen kommen kann - allerdings zulasten der Qualität des Bildes. Neben der mit 50% Kompression gespeicherten Datei PHOTO.JPG werden nachstehend noch JPEG-Dateien mit 60, 70, 80, 90 und (natürlich nur zu Demonstrationszwecken auch mit) 99% Kompression aufgeführt.

```
  Bytes  Dateiname
108.194  PHOTO.JPG
 92.715  PHOTO60.JPG
 76.742  PHOTO70.JPG
 58.580  PHOTO80.JPG
 36.415  PHOTO90.JPG
 13.008  PHOTO99.JPG
```

Welchen Kompressionsgrad man dabei noch akzeptabel findet, hängt von einem selbst ab. Klar ist natürlich, dass bei 99% Kompression von dem ursprünglichen Bild nicht mehr sehr viel übrigbleibt, wie die nachstehenden beiden Schnappschüsse illustrieren sollen, die die 50%- und die 99%-Variante untereinander zeigen.

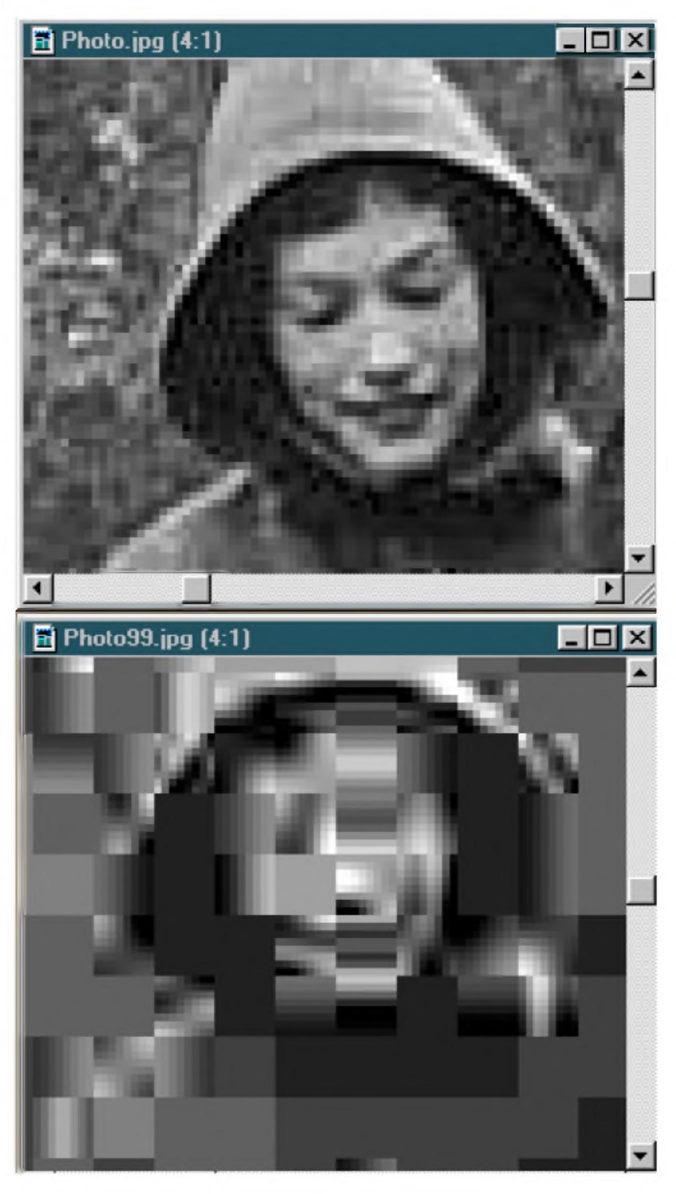


Bild: Vergrößerte Ausschnitte aus zwei JPEG-Dateien
oben: bei 50% Kompression, unten: bei 99% Kompression

1.6. Datenkompression

In diesem Unterkapitel sollen einige wenige Grundlagen der modernen Datenkompression angesprochen werden. Für weitere Lektüre sei u.a. auf das Buch "Grundkurs Informatik" von Hartmut Ernst verwiesen[37].

1.6.1. Ausgangssituation und Begriffsklärung

Unter Kompression versteht man den Vorgang, Daten (z.B. Bild-, Ton- oder Videodaten) so umzukodieren, dass sie weniger Speicherplatz als in ihrem ursprünglichen Format benötigen. Dies kann dazu dienen, dass weniger Platz auf einem Speichermedium erforderlich ist oder eine Datenübertragung (z.B. über das Internet) schneller vonstatten geht.

Anknüpfend an den Begriff der Redundanz (vgl. Def. 1.4.2.4.) können wir formulieren, dass das Ziel der Datenkompression (zunächst) die Reduktion der Redundanz auf (nahe) null ist.

Wir haben im vorherigen Abschnitt bereits eine Anwendung von Datenkompression gesehen: bei der Photo-Abspeicherung kommt es in der Praxis nicht auf eine exakte und damit viel Speicherplatz beanspruchende Darstellung an, es genügt vielmehr, wenn das menschliche Auge das betreffende Bild ohne störende Treppen- oder sonstige Effekte gut erkennen kann.

Damit haben wir ein erstes Merkmal von Kompressionsmethoden kennengelernt: es gibt "exakte" Verfahren, bei denen also - wie bei der Codierung - die ursprüngliche Information in vollem Umfang aus der komprimierten Version wiederhergestellt werden kann. Zum zweiten gibt es "approximative" Verfahren, bei denen - wie bei der oben skizzierten Photo-Darstellung - die ursprüngliche (technische) Information nicht vollständig wieder rekonstruiert werden kann. Hier tritt also ein Verlust an Information auf, der aber für eine bestimmte Art der Anwendung in Kauf genommen wird. Daher spricht man hier von *verlustbehafteten Kompressionsverfahren* im Gegensatz zu den zuerst erwähnten, die entsprechend *verlustfrei* genannt werden.

1.6.2. Verlustfreie Kompressionsverfahren

Will oder muss man die vollständige Ausgangsinformation aus einem komprimierten Datenbestand wiedererhalten, dann ist es zwingend, ein verlustfreies Verfahren einzusetzen. Bei der Mehrzahl der informationstechnischen Datenmengen ist dies nötig - sei es bei Binärcode von ausführbaren Programmen oder Daten einer Datenbank: stets muss man bis auf das letzte Bit die komplette Information zur Verfügung haben.

Bereits in 1.4.2.7. (S. 27) haben wir mit dem Huffman-Algorithmus eine Codierung kennengelernt, die zu einer geringen Redundanz führt. Der daraus resultierende Code heißt *Huffman-Code*.

Aber es geht - zunächst - auch sehr viel einfacher.

---

37 Vgl. [Ernst], S. 90ff.

1.6.2.1. Run Length Encoding

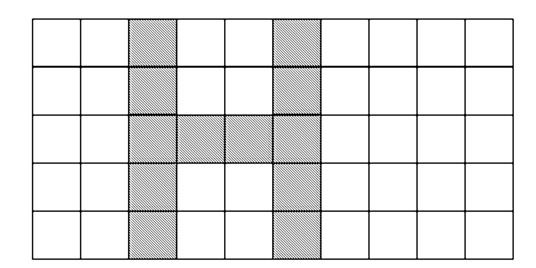
Einer der einfachsten Ansätze einer verlustfreien Kompression ist die andere Darstellung von Wiederholungsstrecken.

So kann eine Sequenz von 25 X-Buchstaben *XXXXXXXXXXXXXXXXXXXXXXXXX* natürlich viel einfacher durch die verkürzte Form *25X* dargestellt werden[38]. Diesen Ansatz nennt man *Lauflängen-Codierung* oder *Run Length Encoding* (*RLE*).

Diese Herangehensweise funktioniert natürlich auch höherdimensional. Eine (in logischer Hinsicht) zweidimensional aufgebaute Pixelgrafik kann entsprechend codiert (komprimiert) werden.

Beispiel: RLE einer Pixelgrafik

In der nachstehenden Skizze sehen Sie den Buchstaben "H" gedanklich als 10 x 5-Pixel-grafik.



Werden schraffierte Felder mit 1, weiße Felder mit 0 codiert, so ist die Grafik darstellbar in der Form[39] "0010010000 0010010000 0011110000 0010010000 0010010000". Hierfür sind also offenbar 50 Bits notwendig.

Mit einer einfachen Lauflängen-Codierung kann dies etwas verkürzt werden, wobei zunächst festgelegt werden muss, ab welcher Wiederholungslänge eine numerische Umcodierung stattfinden soll, denn auch die Darstellung der Zahlen erfordert Speicherplatz.

Exemplarisch nehmen wir für unsere kleine Grafik den Schwellenwert 4. Dann kann (sehr einfach dargestellt) komprimiert werden:

"001001<6:0>1001<6:0><4:1><6:0>1001<6:0>1001<4:0>".

In diesem Beispiel werden also 18 Bits wie zuvor dargestellt, sechsmal tauchen Bitgruppen mit den Vorfaktoren 4 und 6 auf. Abstrahieren wir (aufgrund der hier sehr kleinen Datenmenge) davon, dass auf der Bit-Ebene wieder verwaltungstechnisch geklärt sein muss, wann eine Bit-Sequenz einen numerischen Vorfaktor darstellt und wann die originären Datenbits, dann haben wir folgende Zwischenbilanz: 18 Bits wie zuvor plus 6 Bits für jede der Wiederholungsgruppen zuzüglich sechsmal der Information für einen Vorfaktor.

Das heißt: eine Einsparung in der Datenmenge gegenüber der vorherigen Darstellung haben wir, wenn $18 + 6 + 6 \cdot x < 50$ bzw. $x < 4$ gilt; hierbei ist x der Speicherplatz, den ein Vorfaktor

---

[38] Um präzise zu sein: dies funktioniert wie hier gezeigt natürlich nur dann, wenn die Zifferzeichen (hier '2' und '5') "Meta-Informationen" sind - also selbst nicht zur Ausgangsdatenmenge dazugehören. Andernfalls wäre die Abspeicherung der Information "fünfundzwanzigmal 'X'" komplizierter zu codieren.

[39] Die Leerzeichen wurden hier nur der besseren Lesbarkeit wegen eingefügt.

benötigt. Selbstverständlich macht sich in der Praxis dieser Effekt erst bei größeren Bitmengen positiv bemerkbar.

Der aufmerksame Beobachter hat aber aufgrund des speziellen Bildaufbaus inzwischen sicher eine weitere Idee für eine Kompression bekommen. Anstatt bitweise die Wiederholungssequenzen durchzuzählen kann auch die zweidimensionale Gestalt der Grafik genutzt werden. In der Sequenz “0010010000 0010010000 0011110000 0010010000 0010010000” taucht die erste Zeile (= der erste 10-Bit-Block) viermal auf. Also kann eine Kompression sich auch eine solche Tatsache nutzbar machen, eine Darstellung könnte also lauten: “0010010000 <1> 0011110000 <1> <1>”; dabei bedeutet “<1>” natürlich, dass es sich um eine Kopie der ersten Zeile, des ersten 10-Bit-Blocks handelt. Im Spezialfall wäre das Datenvolumen auf 20+3x Bits geschrumpft, wobei x den Speicherplatz für einen Verweis auf einen 10-Bit-Block bzw. eine Zeile bezeichnet.

Unter dem Stichwort “Quad-Trees” wird ein weiterer Ansatz verstanden, bei dem ein zweidimensionales (rechteckiges) Muster sukzessive in (z.B. jeweils vier) Teilrechtecke zerlegt wird solange, bis ein solches Teilmuster nur noch aus gleichen Bits besteht. (Vgl. [Ernst], S. 91f.)

1.6.2.2. Differenz-Codierung

Für numerische Daten, beispielsweise Messreihen, eignet sich mitunter die sogenannte *Differenz-Codierung*. An Stelle der jeweils “autonomen” kompletten Daten werden nur die Differenzen abgespeichert.

So kann die kleine Sequenz (hier dezimal und mit dem Trennzeichen Komma wiedergegeben) “12837 12839 12766 12822 12825” gemäß einer solchen “Delta-Konvention” auch durch “12837 2 -73 56 3” verlustfrei codiert werden[40].

Verfeinern lässt sich ein solcher Ansatz dadurch, dass bei zu großen Differenzen wieder der vollständige Zahlenwert abgespeichert wird, was umgekehrt jedoch wieder zusätzliche Verwaltungsinformationen voraussetzt.

1.6.2.3. Arithmetische Codierung

Ein weiteres verlustfreies Verfahren ist die sogenannte *Arithmetische Codierung*, die die Häufigkeitsverteilung von Einzelzeichen verwendet. Interessant hierbei der Ansatz: ein kompletter Text wird auf eine einzelne Gleitkommazahl aus dem Intervall [0,1[abgebildet.

Das Verfahren beginnt damit, dass das Intervall [0,1[entsprechend der *n* Zeichen auch in *n* Teilbereiche aufgeteilt wird, deren Breite gerade der relativen Häufigkeit oder Auftrittswahrscheinlichkeit der jeweiligen Zeichen entspricht.

---

[40] Im wissenschaftlichen Bereich wird Ähnliches oft durch eine sog. Datenvorverarbeitung erreicht. In unserem Beispiel hieße das, dass die fraglichen Messdaten zum Beispiel um -12800 oder einen ähnlichen Wert verschoben werden könnten, damit die Werte selbst vergleichsweise klein bleiben.

Bevor wir ein konkretes Beispiel vorstellen, kurz der allgemeine Ablauf des Algorithmus:

Kompressionsalgorithmus:
Ermittle die Intervalle J(z) der einzelnen Zeichen.
Setze unten:=0 und oben:=1 als Startwerte.
Über alle Eingabezeichen z:

```
laenge := oben - unten
oben := unten + laenge * obere_grenze(z)
unten := unten + laenge * untere_grenze(z)
```

Ergebnis: der aktuelle Wert in der Variablen unten.

Die Funktionen obere_grenze() und untere_grenze() geben die jeweiligen Grenzen der Intervalle J(z) des betreffenden Zeichens an.

Die so erhaltenen Ergebnisse sind annähernd gleichverteilt, da jedes Zeichen entsprechend seiner relativen Häufigkeit berücksichtigt wird.

Spielen wir es praktisch durch: zu codieren sei der Text bzw. die Zeichen-Sequenz “ESSEN”. Die relativen Häufigkeiten der drei hier vorkommenden Zeichen sind damit h(‘E’)=0.4, h(‘S’)=0.4 und h(‘N’)=0.2; daraus ergeben sich die drei Intervalle

$$J('E') = [0, 0.4[,\ J('S') = [0.4, 0.8[,\ J('N') = [0.8, 1.0[.$$

Die o.e. Funktionen liefern also z.B. obere_grenze(‘S’) = 0.8 oder untere_grenze(‘S’) = 0.4.

Komprimieren wir das Wort “ESSEN” nun mit dem o.g. Verfahren, dann ergibt sich folgendes Ablaufprotokoll.

	Zeichen z	unten	oben	laenge
Initialisierung		0	1	
	‘E’	0	0.4	1
	‘S’	0.16	0.32	0.4
	‘S’	0.224	0.288	0.16
	‘E’	0.224	0.2496	0.064
	‘N’	0.24448	0.2496	0.0256

Das Ergebnis ist die letzte Belegung der Variablen unten, hier also 0.24448. Das heißt: der Text “ESSEN” wird codiert durch die Gleitkommazahl 0.24448.

Die umgekehrte Operation, die Dekompression, funktioniert wie erwartet:

Dekompressionsalgorithmus:
x := der Code für den fraglichen Text
Solange x > 0
    Suche und Ausgabe des Zeichens z mit $x \in J(z)$
    laenge := obere_grenze(z) - untere_grenze(z)
    x := (x - untere_grenze(z)) / laenge

Im konkreten Zahlenbeispiel ist x = 0.24448. Das Zeichen z mit $x \in J(z)$ ist z = 'E'. Sodann wird die neue laenge berechnet: laenge := 0.4 - 0 = 0.4; anschließend wird x neu gesetzt:
x := (x - untere_grenze('E')) / laenge = (0.24448 - 0)/0.4 = 0.6112.

Dieser Wert ist positiv, also geht es weiter: Das Zeichen z mit $x \in J(z)$ ist nun z = 'S'. Anschließend: laenge := obere_grenze('S') - untere_grenze('S') = 0.4.
x := (x - untere_grenze('S')) / laenge = (0.6112 - 0.4)/0.4 = 0.528.

Auch dieser Wert ist (erwartungsgemäß) positiv, es geht also weiter: Das Zeichen z mit $x \in J(z)$ ist wiederum z = 'S'. Anschließend: laenge := obere_grenze('S') - untere_grenze('S') = 0.4. Und: x := (x - untere_grenze('S')) / laenge = (0.528 - 0.4)/0.4 = 0.32.

Immer noch ist x > 0: Das Zeichen z mit $x \in J(z)$ ist z = 'E'.
laenge := obere_grenze('E') - untere_grenze('E') = 0.4.
x := (x - untere_grenze('E')) / laenge = (0.32 - 0)/0.4 = 0.8.

Auf diese Weise wird jetzt das Zeichen z = 'N' gefunden und ausgegeben. laenge ist danach 0.2, x - untere_grenze('N') = 0.8 - 0.8 = 0. Damit ist der Dekompressionsdurchgang beendet.

Wie man leicht sieht, ist die Arithmetik in diesem Verfahren im Kern auf das Arbeiten mit den vier Grundrechenarten beschränkt. Daher ist für eine exakte Behandlung das Rechnen in der Menge der rationalen Zahlen gut geeignet. Wir können daher die oben protokollierte Codierung des Textes "ESSEN" noch einmal mit Brüchen anstelle von reellen Zahlen dokumentieren.

	Zeichen z	unten	oben	laenge
Initialisierung		0	1	
	'E'	0	2 / 5	1
	'S'	4 / 25	8 / 25	2 / 5
	'S'	28 / 125	36 / 125	4 / 25
	'E'	28 / 125 = 140 / 625	156 / 625 = 28/125 + 8/125*2/5	8 / 125
	'N'	924 / 3125= 140/625 + 56/625 * 4/5	156 / 625 = 28/125 + 16/625*1 = 780 / 3125	16 / 625

Bei der DV-technischen Realisierung kann nun noch dahingehend optimiert werden, dass nur die Potenzen (hier der 5) für den Nenner abgespeichert werden; insbesondere ist die

Darstellung als Bruch stets präzise, während die obige Gleitkommadarstellung rechnerische Ungenauigkeiten mit sich bringt.

1.6.2.4. LZW - Lempel-Ziv-Welch-Algorithmus

Ein sehr effizienter und in der Praxis oft eingesetzter Algorithmus ist das von Lempel und Ziv erfundene und von Welch verbesserte statistische Verfahren, das nicht auf der Basis von Einzelzeichen, sondern unter Berücksichtigung von Wiederholungssequenzen komprimiert.

Der LZW-Algorithmus ist umso effektiver, desto mehr Wiederholungsstücke vorhanden sind - und desto länger diese sind! Das Ergebnis der LZW-Komprimierung ist eine fast redundanzfreie Sequenz, die verlustfrei i.a. nicht mehr weiter komprimiert werden kann.

Das Verfahren arbeitet mit einer Code-Tabelle. In diese werden zu Beginn alle Einzelzeichen des zugrundeliegenden Alphabets mit ihrer entsprechenden Bitsequenz (-dem Code-) eingetragen. Während der Verarbeitung der zu komprimierenden Eingabe wird diese Code-Tabelle um längere Zeichenketten ergänzt; sobald sich diese dann wiederholen, beginnt der positive Effekt der Kompression.

Die Codierung eines Strings S ist nachstehend formuliert.

1. Die Code-Tabelle wird mit allen Einzelzeichen des Alphabets initialisiert.

2. Der Präfix-String P wird auf den leeren String gesetzt, P := “”

3. Für jedes Zeichen z aus der Eingabe S :
    ist Pz in der Code-Tabelle,
        dann P := Pz
    andernfalls
        Eintragen von Pz in die Code-Tabelle
        Ausgabe des Codes für P
        P := z

4. Ausgabe des Codes für P

Auch dieses Verfahren wollen wir kurz an einem konkreten Beispiel illustrieren[41]. Wir komprimieren den String S := “ABABCBABAB”. Als Länge der Code-Tabelle wählen wir (etwas willkürlich) 8, so dass wir mit einem 3-Bit-Code auskommen.

Die Initialisierung der Code-Tabelle sieht also wie folgt aus.

[41] Das numerische Beispiel ist aus [Ernst], S. 103f entnommen.

Präfix	Code
A	000
B	001
C	010
	011
	100
	101
	110
	111

Nun wird der String S codiert. Das Präfix P wird auf den leeren String “” gesetzt. Gelesen wird z := ‘A’.

akt. Position	Präfix P	Eintrag in die Code-Tabelle	Ausgabe
ABABCBABAB		(init.)	
ABABCBABAB	A		
ABABCBABAB	B	AB = 011	000 (Code für A)
ABABCBABAB	A	BA = 100	001 (Code für B)
ABABCBABAB	AB		
ABABCBABAB	C	ABC = 101	011 (= AB)
ABABCBABAB	B	CB = 110	010 (= C)
ABABCBABAB	BA		
ABABCBABAB	B	BAB = 111	100 (= BA)
ABABCBABAB	BA		
ABABCBABAB	BAB		
ABABCBABAB			111 (= BAB)

Anschließend sieht die Code-Tabelle wie folgt aus.

Präfix	Code
A	000
B	001
C	010
AB	011
BA	100
ABC	101
CB	110
BAB	111

Die generierte Ausgabe lautet somit 000 001 011 010 100 111 (oder dezimal notiert zur besseren Lesbarkeit) 013247.

Die LZW-Dekomprimierung funktioniert entsprechend "rückwärts":

1. Die Code-Tabelle wird mit den Einzelzeichen initialisiert (wie zuvor).

2. P := ""

3. Für jedes Eingabezeichen (bzw. jede Eingabebitgruppe) c :
    Ist c in der Code-Tabelle enthalten?
    Falls ja:
        Ausgabe des Strings zum Code c
        z := 1.Zeichen dieses Strings
        Eintragen von Pz in die Code-Tabelle (falls noch nicht vorhanden)
        P := String zum Code c
    Falls nicht:
        z := 1.Zeichen von P
        Ausgabe Pz
        Eintragen von Pz in die Code-Tabelle
        P := Pz

Auch bei diesem Algorithmus bieten sich eine Reihe von Optimierungen in der praktischen Umsetzung an; beispielsweise wird eine Code-Tabelle in der Praxis deutlich mehr als acht Einträge aufweisen, so dass sich für das Suchen und Auffinden von Einträgen eine Hash-Codierung anbietet.

1.6.3. Verlustbehaftete Datenkompression

Bei verlustbehafteten Kompressionsverfahren geht ein Teil der Originalinformation (z.B. der Grafik) verloren. Hat etwa eine Anzahl von Pixeln nur minimal verschiedene Farbwerte, so kann hier „gerundet" ein einziger Farbwert verwendet werden. Diese Pixel bilden dann eine gleichfarbige Gruppe. Im Idealfall ist der Unterschied zum Original für das menschliche Auge nicht sichtbar. Im Vergleich zu nicht-verlustbehafteten Verfahren sind mit verlustbehafteter Kompression naturgemäß wesentlich höhere Kompressionsraten erreichbar.

Auch hier zunächst ein einfaches Beispiel einer verlustbehafteten Komprimierung.

1.6.3.1. Verlustbehaftete Differenz-Codierung

Die im vorigen Abschnitt besprochene Differenz-Codierung ist naheliegenderweise auch verlustbehaftet möglich. Die Sequenz "12837 12839 12766 12822 12825" kann, wenn es nur auf Größenordnungen modulo 10 ankommt, als "12840 0 -70 60 0" gespeichert werden. Selbstverständlich muss hier über lange Strecken mit kumulierten Rundungsfehlern gerechnet werden, so dass sich "von Zeit zu Zeit" eine Nachjustierung anbietet, d.h. in gewissen Abständen sollte hier wieder ein Originalwert anstelle einer Differenz gespeichert werden.

1.6.3.2. JPEG und MPEG

Der bekannteste verlustbehaftete Kompressionsalgorithmus ist *JPEG* (*Joint Photographic Experts Group*). Hierzu gehört das gleichnamige Dateiformat, das in der Windows- und DOS-Welt üblicherweise mit der Extension .jpg einhergeht. Auch das Video-Format *MPEG* (benannt nach der *Moving Pictures Expert Group*) ist ein Beispiel für ein verlustbehaftetes Kompressionsverfahren.

Der JPEG-Standard umfasst eine Reihe von grundlegenden Kompressionsmethoden, wobei sich für die Bildcodierung das auf der sog. diskreten Kosinustransformation (*discrete cosine transformation*, *DCT*) beruhende Verfahren durchgesetzt hat[42].

Aus Zeitgründen wird auf die mathematische Behandlung der Fourier-Transformation u.a. an dieser Stelle verzichtet. Stattdessen sei stellvertretend auf die recht umfassende Ausführung von B. L. Winkler (sh. [Winkler] im Literaturverzeichnis) verwiesen, die im FHDW Intranet auf der Startseite zu "Grundlagen der Informatik" als PDF-Datei heruntergeladen werden kann.

[42] Siehe hierzu z.B. die Ausführungen von Joachim Schwarz und Guido Sörmann, [SchwSör] im Literaturverzeichnis.

2. SOFTWARE

Der Betrieb eines Computers setzt, vereinfacht gesagt, funktionierende Hardware und Software voraus. Salopp formuliert ist Hardware das, was man prinzipiell anfassen kann (siehe Seiten 63 ff). Software ist[43] der Sammelbegriff für die System- und die Anwendungsprogramme einer Datenverarbeitungsanlage. Ein Programm ist dabei eine endliche Folge von Befehlen, die der Computer, eventuell wiederholt, abzuarbeiten hat.

Die gesamten, aufeinander abgestimmten Systemprogramme bilden gemeinsam das Betriebssystem des Rechners. Dieses steuert und überwacht den Ablauf von Anwendungsprogrammen.

Man unterscheidet hierbei zwischen Single- und Multi-User-Systemen, je nachdem, wieviele (logische) Benutzer gleichzeitig mit dem System arbeiten können, und zwischen Single- und Multi-Processing-Systemen. Bei einem Multi-Processing-System kann nicht nur ein Prozess zu einer Zeit abgearbeitet werden, sondern es teilen sich ggf. viele Prozesse die zur Verfügung stehende Rechenzeit.

Zusätzlich gibt es systemnahe Software, die das Funktionsspektrum des Betriebssystems selbst übertrifft, aber noch unterhalb der eigentlichen Anwenderschicht anzusiedeln ist. Hierzu gehören Datenbankmanagementsysteme (DBMS), Datenfernverarbeitungssoftware oder Programmentwicklungssysteme.

Anwendungsprogramme bieten dagegen Lösungshilfen für fachliche Probleme. Dazu gehören technisch-wissenschaftliche, kommerzielle, auf betriebliche Funktionen bezogene Programme sowie spezielle Branchensoftware. Auch die bekannten, eher universell einsetzbaren Textverarbeitungs- und Tabellenkalkulations- oder Graphikprogramme gehören in diese Rubrik.

2.1. Programmdefinition

Ein *Algorithmus* wird[44] definiert als „endliche Folge eindeutiger Anweisungen zur Lösung eines Problems". Unter einem *Programm* versteht man einen „Algorithmus in einer der Maschine übermittelbaren Form", also eine Verarbeitungsvorschrift bestehend aus einer endlichen Folge von einzelnen Instruktionen, die im Maschinencode des jeweiligen Rechners oder in einer noch in den entsprechenden Maschinencode zu übersetzenden Sprache vorliegen. Hierbei werden Daten manipuliert, die in sogenannten Datenstrukturen (Datentypen) logisch aufgebaut und physisch im Speicher[45] des Rechners gehalten werden.

Nach Niklaus Wirth[46] ergibt sich die klassische „Kurzdefinition"[47]:

$$\textit{Programm} = \textit{Algorithmus} + \textit{Daten(strukturen)}$$

---

[43] Zitiert nach [Hansen].

[44] gemäß DIN 44 300

[45] Dabei ist es zunächst unerheblich, ob es sich hierbei um physischen Arbeitsspeicher oder eine andere Form von Speicher handelt.

[46] Siehe [Wirth].

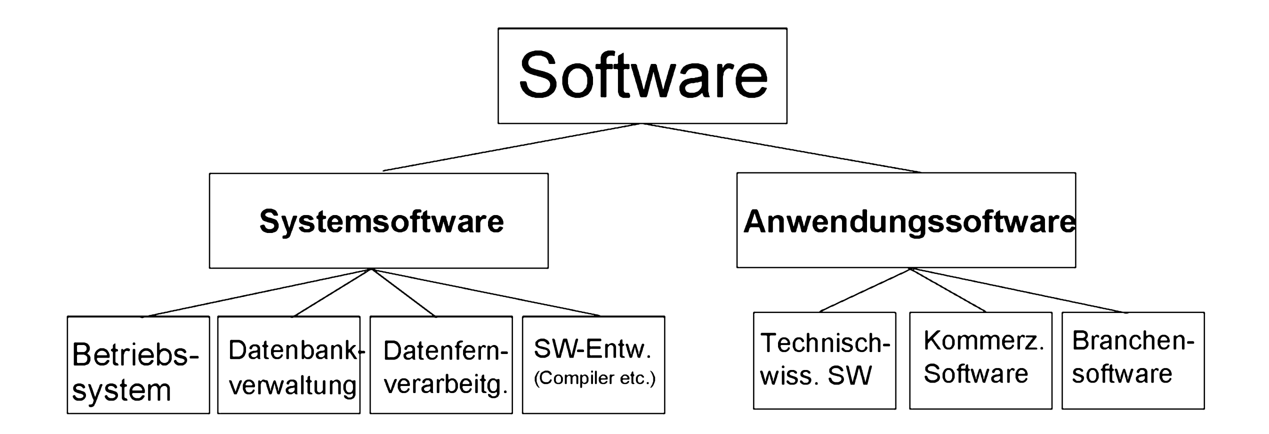
[47] Hier zitiert nach [Lehner], S. 313.

Ein solches Programm wird konkret formuliert in einer Programmiersprache, laut DIN 44 300 ein „Mittel zur eindeutigen Formulierung eines Algorithmus", beispielsweise C, Java, Pascal, COBOL. Auf die verschiedenen Programmiersprachen wird später (in Abschnitt 2.3.) genauer eingegangen.

Ein in einer solchen Sprache formulierter Code, z.B. in Pascal, muss durch einen Compiler übersetzt werden in vom Prozessor interpretierbaren Maschinencode.
Gemäß DIN 44 300: Ein *Compiler* ist ein (Übersetzer-)Programm zur Umformung eines Programmiersprachen-Programms (Quellcode, source code) in ein Maschinenprogramm (Binärcode, object code.

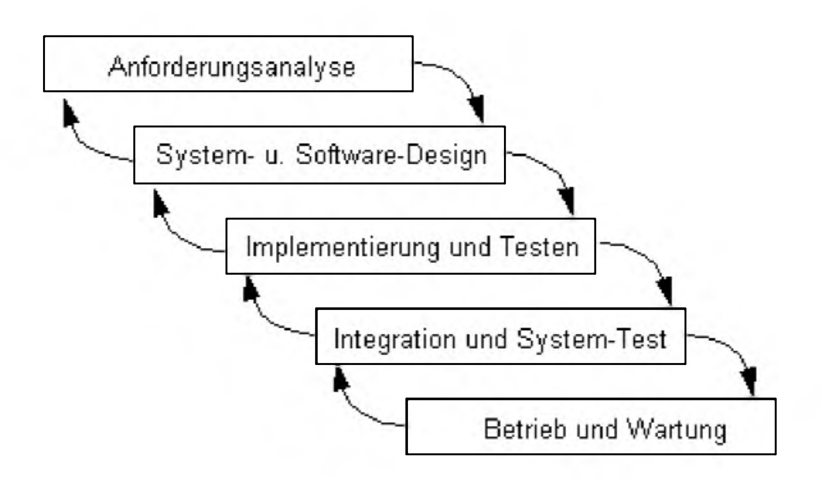
Im Bereich der Software unterscheidet man häufig zwischen der sog. System- und der Anwendungssoftware.

Software	Sammelbegriff für die →Systemprogramme und die →Anwendungsprogramme einer EDV-Anlage
Systemprogramme	Die gesamten, aufeinander abgestimmten Systemprogramme bilden das Betriebssystem eines Rechners. Es steuert und überwacht die Abwicklung von Anwendungsprogrammen. Insbesondere hat das Betriebssystem die sog. Betriebsmittel zu verwalten: Prozessoren, Prozesse (=laufende Programme), Arbeitsspeicher, Ein-/Ausgabemedien, Dateien u.a. Daten.
Anwendungsprogramme	Die Anwendungsprogramme bieten Lösungen für fachliche Problemstellungen; dazu gehören neben den Standardprogrammsparten Textverarbeitung, Tabellenkalkulation, Datenbankapplikation, Terminkalender usw. auch technisch-wissenschaftliche Programme (z.B. für mathematische oder statistische Berechnungen), kommerzielle / allgemein betriebliche Programme (etwa für die Finanzbuchhaltung) und spezielle Branchenprogramme.



2.2. Software- und Programmentwicklung

Unter *Software* versteht man die Gesamtheit aller Programme, die auf einem DV-System vorhanden sind. Software unterliegt dem allgemeinen Lebenszyklus von Anwendungssystemen, dem sogenannten Software-Lebenszyklus: die nachstehende Skizze zeigt das sogenannte *Wasserfallmodell*.



1. **Anforderungsanalyse**
   Welche Aufgaben sind zu erfüllen?
2. **System- und Software-Design**
   Entwurf des erforderlichen Gesamtsystems und der Software
3. **Implementierung und Testen**
   Erstellen und Testen der Software
4. **Integration und Systemtest**
   Einbinden der Software in den betrieblichen Ablauf und Testen der Software im Gesamtrahmen
5. **Betrieb und Wartung**
   Produktiver Einsatz der Software mit parallel stattfindenden Wartungsarbeiten

Die obige Skizze soll deutlich machen, dass es sich hierbei um keine Einbahnstraße in der Software-Entwicklung handelt: vielmehr kann aus jeder Phase auch wieder eine Rückwirkung auf die Phase(n) davor erfolgen, die eine Modifikation des konkreten Programmocdes oder sogar des Designs der Software erforderlich machen kann.

Neben dem hier gezeigten Wasserfallmodell gibt es natürlich noch andere Veranschaulichungen des Grundproblems Software-Lebenszyklus; andere Modelle betonen, dass es sich nicht um streng abgeschlossene Phasen (wie hier beim Wasserfallmodell) handelt.

Zitiert nach [Stahlknecht] stellt sich die Systementwicklung in einem Unternehmen skizzenartig dar wie folgt.

Unter dem Stichwort *Realisierung* werden in der Regel die Phasen Programmentwicklung (Programmierung, Codierung) und der Programm- und Systemtest zusammengefasst. Gelegentlich trennt man Programmentwicklung von Codierung: dann meint ersteres das „Denken", das Entwickeln der Algorithmen und deren Umsetzung in die Programmiersprache, und das Codieren ist das „Tippen", das konkret „mechanische" Erzeugen des Programmcodes.

Zur Dokumentation und ggf. leichteren Lesbarkeit von Algorithmen wurden verschiedene Methoden entwickelt. Gab es früher einmal *Programmablaufpläne* (*PAP*), so sind es heute - neben der Dokumentation in einem Pseudo-Code - in der Regel *Struktogramme* (nach Nassi-Shneidermann), mit denen Programmabläufe dargestellt werden.

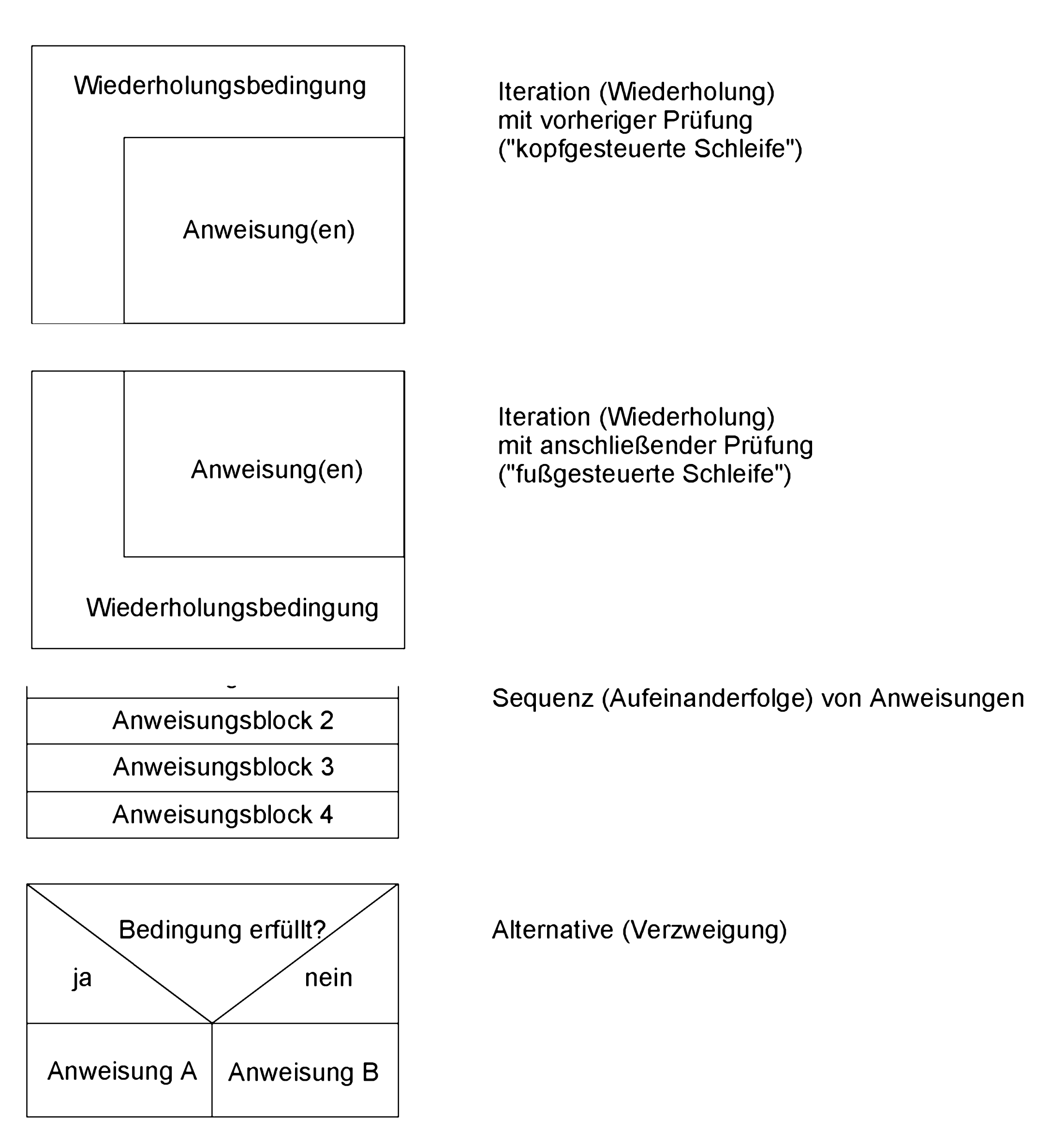


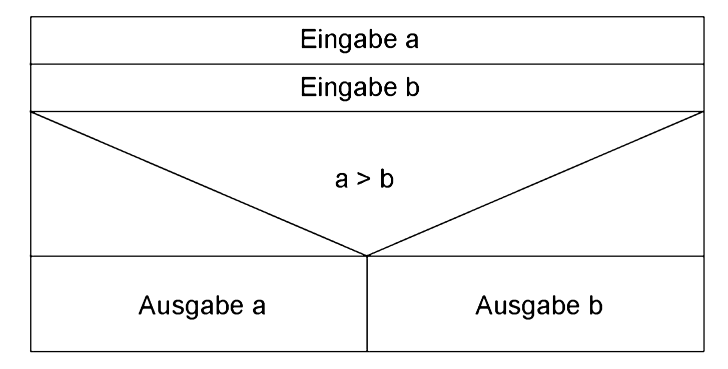
Bild: Verschiedene Struktogramme nach Nassi-Shneidermann

Für den Pseudo-Code wird häufig die Programmiersprache Pascal verwendet (oder zumindest ein Pseudo-Code angelehnt an Pascal benutzt). Der Einfachheit halber wollen wir uns hier ein wenig Pascal[48] vornehmen.

Beispiel 1: Die größere von zwei zunächst einzugebenden ganzen Zahlen soll auf den Bildschirm ausgegeben werden. Hierfür werden zwei Speicherplätze für ganze Zahlen (Englisch: integer) benötigt, die wir hier a und b nennen wollen.

Der (Mini-)Algorithmus für dieses Beispiel lautet dann:
1. Eingeben von a und b von Tastatur
2. Wenn a größer als b ist (mathematisch: a > b), dann wird a ausgegeben, sonst b

In einem Struktogramm sieht dies so aus.



In unserem Pseudo-Code Pascal lässt sich dieses Beispiel wie folgt formulieren.

```
program beispiel1;
var a, b: integer;
begin
read(a);
read(b);
if a > b then
   write(a)
else
   write(b)
end.
```

Zunächst wird hier gemäß der Syntax von Pascal festgehalten, dass es sich um das Programm namens „beispiel1” handelt. Dann wird notiert, welche Variablen (Speicherplätze) benötigt werden und von welchem Datentyp[49] diese sind. Mit „begin” und „end” werden die Anweisungen des Pascal-Programms eingerahmt.

Zunächst werden mit den read-Anweisungen die Variablen a und b von Tastatur eingelesen; anschließend wird durch die Bedingung „wenn a größer als b ist” verzweigt, je nachdem, welche Werte in a und b aktuell stehen. Ist beispielsweise a=1 und b=2, so trifft die

[48] In der PC-Welt stammt der dominierende Pascal-Dialekt von der Firma Inprise (vormals Borland), die derzeit unter dem Produktnamen Delphi den Nachfolger des legendären Turbo Pascal vertreibt.

[49] Der Datentyp legt fest, was in diesen Variablen gespeichert werden soll. integer steht für „ganze Zahlen”; gleichzeitig ist hiermit auch bestimmt, wieviel Speicherplatz vom Compiler bereitgestellt werden soll. Beim Turbo Pascal Compiler von Borland erhält eine integer-Variable zwei Byte Speicher bereitgestellt. Eine solche Variable kann also, wie man leicht nachrechnet, (zu verschiedenen Zeitpunkten) 65536 verschiedene Werte beinhalten. In der Regel ist dies dann der Zahlenbereich von -32768 bis +32767.

Bedingung nicht zu - und das Programm geht zum else-Zweig, gibt hier also den Wert von b, die 2, auf den Bildschirm aus.

2.3. Programmiersprachen in der Übersicht

Man kann Programmiersprachen nach verschiedenen Gesichtspunkten kategorisieren. Eine Sichtweise ist das Sprachniveau, d.h. in welcher Nähe zum Prozessor ist der Programmcode zu formulieren.

Maschinenorientierte Sprachen	Maschinensprachen	prozessorspezifisch, z.B. 80x86- oder 8088-Maschinencode *0001 1010 0011 0100* „Addiere" 3 4
	Assembler	IBM PC-Assembler *ADD 3,4*
Problemorientierte, höhere Programmiersprachen	Allgemeine höhere Sprachen (all purpose languages) - *prozedural* (Pascal, C) - *objektorientiert* (C++, Java, Smalltalk)	BASIC (Beginners All Purpose Symbolic Instruction Code) *LET SUMME = 3 + 4* Pascal *summe := 3+ 4* C/C++/Java *summe = 3+4;* COBOL *MOVE 3 + 4 TO SUMME*
	Spezialisierte höhere Sprachen	aus dem Bereich der Künstlichen Intelligenz (KI): ProLog (Programmieren in Logik) LisP (Listenprogrammierung) SNOBOL (Zeichenkettenverarbeitung)

Nachfolgend wird ein Überblick über die verschiedenen Generationen von Programmiersprachen gegeben. Er basiert auf einem Text der Universität Halle, der unter der Internet-Adresse http://www.informatik.uni-halle.de/lehre/c/c12.html abgerufen werden kann.

Programmiersprache: Sprache zur Formulierung von Rechenvorschriften, d.h. von Datenstrukturen und Algorithmen, die von einem Computer ausgeführt werden können. Häufig werden heute 5 Generationen von Programmiersprachen unterschieden:

1. Generation: Maschinensprachen

Befehle werden direkt in einer Maschinensprache notiert, d.h. als Folge von Zahlencodes. Da sich der Befehlssatz von Rechner mit unterschiedlichen Prozessoren im allgemeinen unterscheidet, sind in Maschinensprache geschriebene Programme nur sehr schwer

übertragbar. Die direkte Programmierung in einer Maschinensprache wird heute kaum noch verwendet.

Einige Programmiersysteme für höhere Programmiersprachen gestatten es, Maschinenbefehle in den Quelltext zu integrieren. Die Anwendung beschränkt sich dann auf solche Fälle, in denen es aus funktionalen oder Effektivitätsgründen unumgänglich oder nützlich ist, maschinennah zu programmieren.

2. Generation: Assembler

Anstelle von Zahlencodes wird mit Hilfe von symbolischen Bezeichnern codiert. Eine Assembleranweisung wird in genau einen Maschinenbefehl umgesetzt. Auch Assemblerprogramme sind deshalb im allgemeinen an einen bestimmten Prozessortyp gebunden.
Makroassembler gestatten die Bildung von parametrisierbaren Befehlsgruppen. Eine Makroanweisung wird im allgemeinen in mehr als einen Maschinenbefehl umgesetzt.

Der Anteil der Assemblerprogrammierung ist im Sinken. Der Möglichkeit der Erstellung effektiver Programme steht die erschwerte Wartbarkeit von Assemblerprogrammen gegenüber. Maschinennahe Programmierung - die Domäne von Assembler - kann heute überwiegend durch höhere Programmiersprachen abgedeckt werden. Hierfür kommt z.B. C in Frage, auf dem PC zum Teil auch Turbo Pascal.

Einige Programmiersysteme für höhere Programmiersprachen gestatten es, Assemblerbefehle in den Quelltext zu integrieren. Die Anwendung kann sich dann auf die Situationen beschränken, in denen es aus funktionalen oder Effektivitätsgründen notwendig oder nützlich ist, maschinennah zu programmieren.

3. Generation: höhere Programmiersprachen (high level language)

Sprachen der 3. Generation unterstützen unmittelbar die Notation von Algorithmen, sie sind weitgehend anwendungsneutral und maschinenunabhängig.
Erste höhere Programmiersprachen entstanden ab Mitte der fünfziger Jahre (ForTran, COBOL, ALGOL-60). Weitere Sprachen dieser Generation sind zum Beispiel Pascal, Modula-2, PL1, C, ADA, BASIC, SIMULA.

4. Generation: Fourth Generation Language (4GL)

Sprachen der 4. Generation sind anwendungsbezogen (applikative Sprachen). Sie stellen i.a. die wichtigsten Gestaltungsmittel von Sprachen der 3. Generation zur Verfügung, zusätzlich jedoch Sprachmittel zur Auslösung von relativ komplexen, anwendungsbezogenen Operationen, beispielsweise zum Zugriff auf Datenbanken und zur Gestaltung von Benutzeroberflächen.

Sprachen der 4. Generation gehören häufig zum Umfeld von Datenbanksystemen. Eine relativ weit verbreitete Sprache dieser Art ist z.B. *NATURAL* von Adabas oder *SQL* (structured query language, strukturierte Abfragesprache).

5. Generation: (Very High Level Language, VHLL)

Sprachen der 5.Generation gestatten das Beschreiben von Sachverhalten, von Problemen. Sie kommen vor allem im Bereich der KI (künstliche Intelligenz) zum Einsatz. Die Wahl des

Problemlösungsweges kann (entsprechend dem Sprachkonzept) dem jeweiligen System (weitgehend) überlassen werden.

Bekanntestes Beispiel für eine Sprache der 5. Generation ist ProLog (Programmieren in Logik).

Sprachen der 1. Generation sind zwangsläufig hardwareabhängig. Auch Sprachen der 2. Generation sind stark hardwarebezogen. Sprachen beider Generationen werden daher auch als maschinenorientierte Sprachen bezeichnet.

Mit den Sprachen der 3. bis 5. Generation wird eine weitgehende Hardwareunabhängigkeit angestrebt. Die Darstellung der Problemlösung (3. Generation) bzw. des Problems selbst (4. und vor allem 5. Generation) rückt stärker in den Mittelpunkt. Diese Sprachen lassen sich auch als problemorientierte Sprachen kennzeichnen.

Gegenwärtig spielen vor allem die nachstehend genannten Programmiersprachen eine Rolle, die alle als prozedural oder objektorientiert einzustufen sind (die Reihenfolge ist alphabetisch):

- Ada
  Ergebnis einer Ausschreibung des USA-Verteidigungsministeriums in der zweiten Hälfte der siebziger Jahre
  gute Standardisierung (1979, 1983), neuer Standard Ada 95
  als universelle Programmiersprache konzipiert, sehr großer Sprachumfang

- BASIC (Beginners All-purpose Symbolic Instruction Code)
  als Anfänger-Programmiersprache Mitte der sechziger Jahre konzipiert (T. Kurtz, J. Kemeny)
  BASIC-Systeme arbeiten häufig interpretativ
  neuere BASIC-Systeme (z.B. Visual BASIC) bieten häufig eine große Leistungsbreite (strukturierte Programmierung, Grafik, Datenbankzugriff), gewährleisten jedoch keine Portabilität

- **C**
  wurde Anfang der siebziger Jahre in den AT&T Bell Laboratories als Systemprogrammiersprache im Zusammenhang mit dem Betriebssystem UNIX entwickelt (D. Ritchie)
  heute auf allen Rechner- und Systemplattformen verfügbar, gute Standardisierung
  wachsende Bedeutung im Bereich der professionellen Software-Entwicklung
  großer Gestaltungsspielraum - im positiven wie im negativen - für den Programmierer, deshalb wenig geeignet als Ausbildungssprache

- **C++**
  Erweiterung von C um Ausdrucksmittel der objektorientierten Programmierung Anfang der achtziger Jahre in den AT&T Bell Laboratories (B. Stroustrup), Sprachstandardisierung in der Endphase (1997 abgeschlossen)

- C#
  Unter dem Namen “C#” - lies: *c-sharp* - hat Microsoft wieder einmal einen Versuch unternommen, offene und anerkannte Standards um selbstgebastelte Varianten zu

erweitern. C# ist im Kern ein aus Java und C++ zusammengerührte Programmiersprache, die im Kontext von "*.NET*" - lies *dot-net* - eingesetzt wird.

- COBOL (Common Business Oriented Language)
  war (und ist teilweise noch) weit verbreitet im Bereich der kommerziellen Datenverarbeitung,
  für technische Aufgabenstellungen wenig geeignet,
  entstand Ende der Fünfziger Jahre des vorigen Jahrtausends (CODASYL Commitee),
  Weiterentwicklung über verschiedene Standards: COBOL 74, COBOL 85, Object COBOL.

- ForTran (Formula Translator)
  universelle Programmiersprache, vor allem geeignet für den wissenschaftlich-technischen Bereich, erste Version Mitte der Fünfziger Jahre entwickelt (J. Backus),
  Weiterentwicklung der Sprache über verschiedene Standards: FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95.

- **Java**
  aus C++ von der Firma Sun entwickelte objektorientierte Programmiersprache ursprünglich insbesondere zur Programmierung von Chips in Geräten (Waschmaschine, Kaffeeautomat usw.), dann Durchbruch in Zusammenhang mit der Programmierung im Internet und Intranet (Programme und Applets).

- Pascal
  universelle Programmiersprache; Ende der sechziger, Anfang der siebziger Jahre speziell für Ausbildungszwecke entwickelt (Niklaus Wirth, ETH Zürich), verbreitet vor allem für die Entwicklung von Individualsoftware auf PC, spielt im Bereich der professionellen Software-Entwicklung nur eine geringe Rolle.
  Programmierung „im Großen" wird durch den ursprünglichen PASCAL-Standard (1982) nicht unterstützt.

- PL1 oder PL/I (Programming Language 1)
  Mitte der Sechziger Jahre bei IBM entwickelte universelle Programmiersprache - sie vereinigt Elemente von ALGOL-60, COBOL und Fortran und besitzt einen sehr großen Sprachumfang. Allerdings hat PL/I über den Mainframe-Bereich hinaus kaum Bedeutung erlangt.

3. HARDWARE

Der Aufbau eines modernen Rechners[50] nach dem sogenannten von-Neumann-Prinzip lässt sich schematisch wie folgt darstellen.

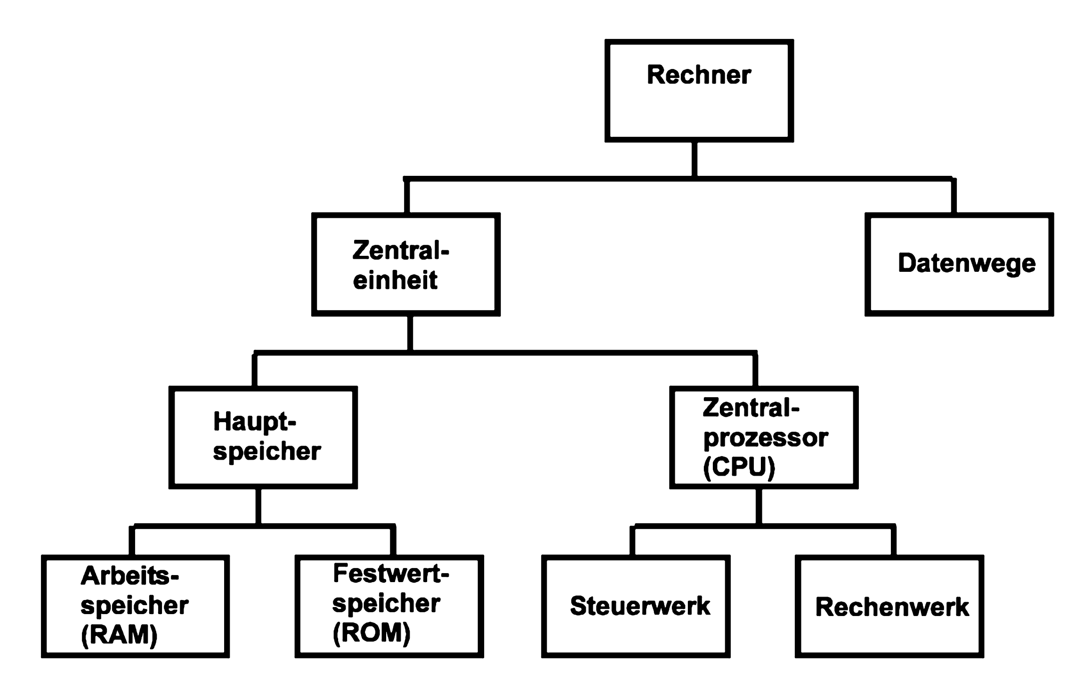
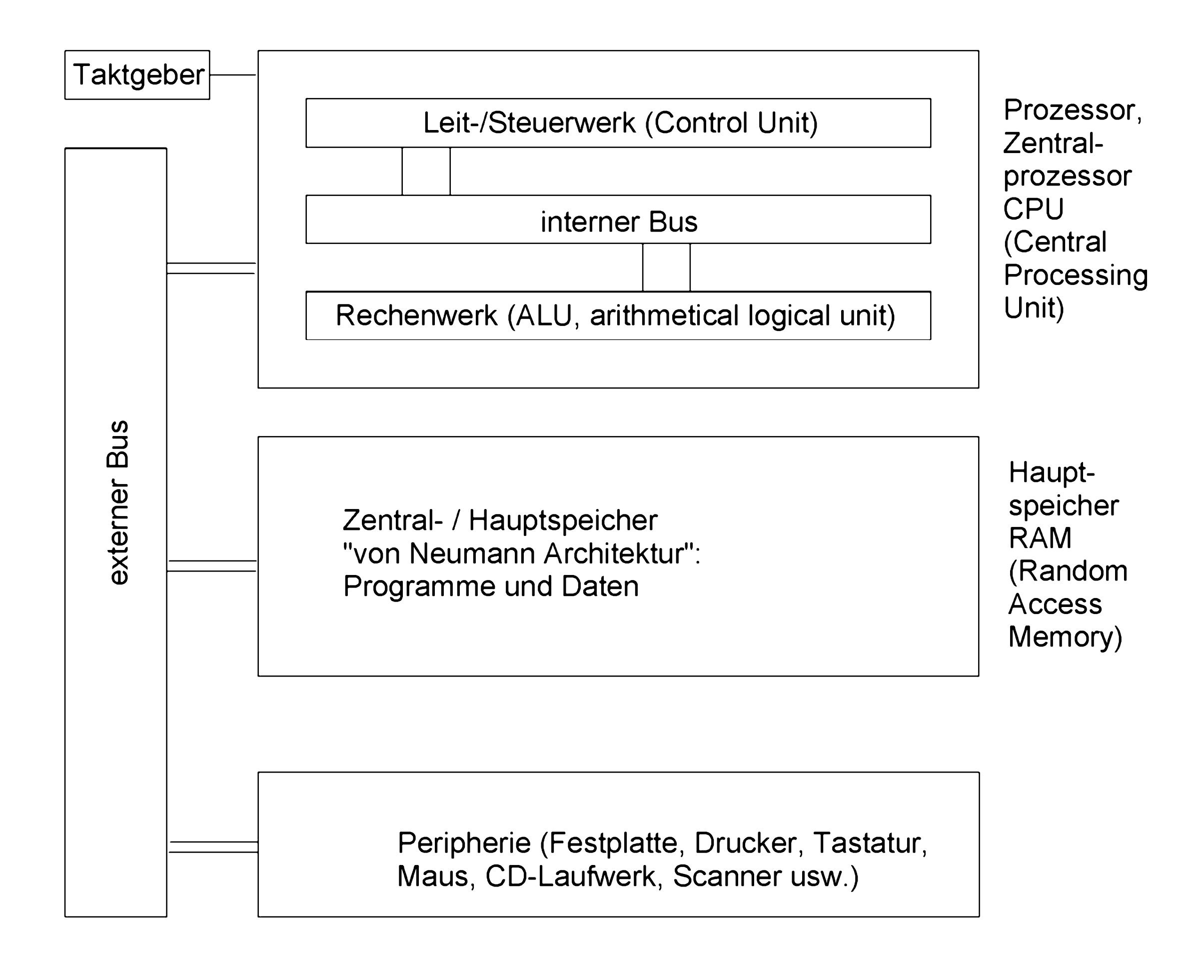


Bild: Schema der von-Neumann-Rechnerarchitektur

[50] Meist stellen wir uns heutzutage darunter einen PC (Personal Computer) oder höchstens noch einen Apple Macintosh („Mac") vor!



Schließlich wird ebenfalls über den externen Bus die gesamte Peripherie angesprochen: dazu gehören auch in das PC-Gehäuse eingebaute Komponenten wie Festplatten oder die Graphikkarte, aber auch gesondert angeschlossene Geräte wie der Monitor, die Tastatur, die geliebte Maus, der Drucker, das Modem, der Scanner u.v.a.m.

3.2. Datenein- und -ausgabe

Maßgeblich für die Kommunikation zwischen Mensch und Computer sind die zur Verfügung stehenden Möglichkeiten der Datenein- und -ausgabe, auf die in diesem Abschnitt eingegangen werden soll.

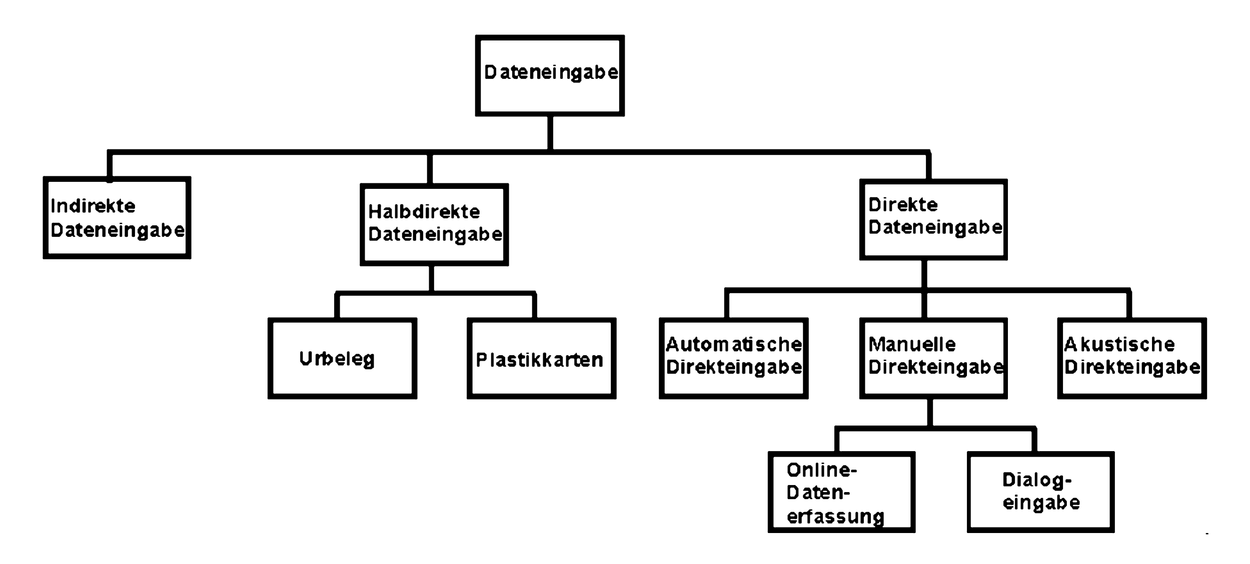
3.2.1. Dateneingabe

Die Dateneingabe in die DV-Anlage steht am Anfang des EVA-Prinzips[51]. Die Dateneingabe erfolgt

☞ indirekt über Datenträger,

☞ halbdirekt vom Urbeleg (Originalbeleg) oder von Identifikationskarten oder

☞ direkt, und zwar automatisch, manuell oder akustisch.



3.2.1.1. Indirekte Dateneingabe

Bei der indirekten Dateneingabe geht der eigentlichen Dateneingabe eine manuelle Datenerfassung auf Datenträger voraus und zwar auf gelochte Datenträger (Lochkarte, Lochstreifen) oder auf magnetische Datenträger (Diskette, Magnetband, Magnetband-kassette).

3.2.1.2. Halbdirekte Dateneingabe

☞ die Daten werden von Urbelegen, auf den sie als Markierung (z. B. EAN) oder in Form von Handblock- oder Maschinenschrift aufgezeichnet sind, mit Lesegeräten unmittelbar in die DV-Anlage eingelesen

☞ die Daten werden mit Plastikkarten (Scheck-, Ausweis- oder Krankenversicherungskarte) auf denen sie auf einem Magnetstreifen oder ein Chip codiert sind, mit spezielle Lesegeräten in die DV-Anlage eingelesen.

3.2.1.3. Automatische Direkteingabe

Eingabewerte werden von Sensoren erfasst und direkt an die DV-Anlage weitergeleitet (PDE = Prozeßdatenerfassung).

---

[51] EVA: Eingabe - Verarbeitung - Ausgabe

3.2.1.4. Manuelle Direkteingabe

Manuelle Direkteingabe erfolgt

☞ mit Tastaturen (ggfs. mit Zusatzgeräten wie Maus, Trackball etc),

☞ mit dem Lichtstift am graphischen Bildschirm oder

☞ durch Berührung mit dem Finger an Touchscreens.

3.2.1.5. Akustische Direkteingabe (Spracheingabe)

Die akustische Direkteingabe (Spracheingabe) erfolgt über Mikrophone möglicherweise in Verbindung mit einer Funkübertragung. Der DV-Anlage müssen Geräte zur Sprachumwandlung vorgeschaltet sein.

3.2.1.6. Wirtschaftlichkeit der Dateneingabe

Das Thema Wirtschaftlichkeit der Dateneingabe wird in der Praxis häufig unterschätzt. Wenn die Daten dort wo sie anfallen, eingegeben werden sollen, setzt dieses möglicherweise Standleitungen. Deshalb ist Dateneingaben immer unter dem Aspekt der Wirtschaftlichkeit zu betrachten, wobei Dateneingabe und Datenausgabe zusammen betrachtete werden müssen. Hinsichtlich der Dateneingabe sind folgende Fragen zu stellen:

☞ Wo fallen die Daten an?
Beispiele: Büro, Werk, Kasse, Schalter; zentral, dezentral, ortsfest/mobil.

☞ Wie fallen die Daten (gegenwärtig) an?
Beispiele: Originalbeleg, z. B. Eingangsrechnungen, Durchschläge, z. B. bei Ausgangsrechnungen, spezielle Erfassungsbelege, beleglos, etwa bei Scheckkarteneingabe oder Tastaturbedienung;

☞ Wann und wie oft fallen die Daten an?
Beispiele: periodisch, unregelmäßig, auf Anforderung;

☞ Wie viele Daten fallen an
Beispiele: hoher/niedriger, gleichmäßiger/ungleichmäßiger Anfall, Spitzenbelastungen.

☞ Wie schnell müssen welche Daten wo verarbeitet werden und wo zur Verfügung stehen?
Beispiele: Lagerbestände beim Einkauf, Bestellungen beim Vertrieb, Gehaltsnachweise beim Mitarbeiter, Kennzahlen bei der Geschäftsleitung.

☞ Wo und wie oft müssen Daten aktualisiert werden („gepflegt”) werden.

Überlicherweise werden folgende Ziele verfolgt:

☞ Vermeidung von Mehrfacherfassung derselben Daten. Lösungen bieten der Datenträgeraustausch und der elektronische Datenaustausch, auch zwischen Unternehmen.

☞ Reduzierung des Beleganfalls durch andere Eingabetechniken reduziert. Typische Beispiele hierfür sind Krankenversicherungskarten, Elektronic Cash, Telebanking, Direkteingabe mit Notepads (z. B. bei UPS), und die Automatisierung von Erfassungsvorgängen, beispielsweise bei computergestützen Lagerverwaltungsssytemen.

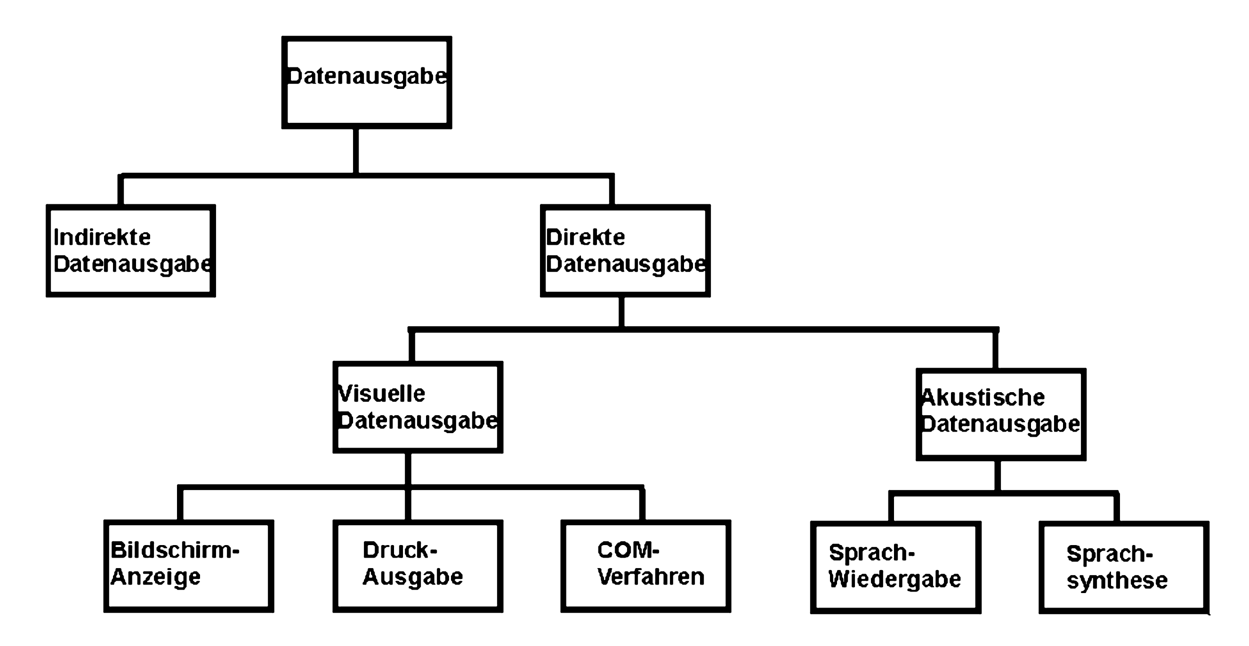
3.2.2. Datenausgabe

Die Ausgabe von Daten kann

☞ indirekt, d. h. in (nur) maschinell lesbarer Form oder

☞ direkt, d. h. in visueller Form oder in akustischer Form

erfolgen.



3.2.2.1. Indirekte Datenausgabe

Die Ausgabe in maschinell lesbarer Form dient der Zwischenspeicherung

- für eine spätere Weiterverarbeitung oder
- für eine spätere Datenausgabe in visuell lesbarer Form (meistens auf einem Drucker).

3.2.2.2. Direkte Datenausgabe

Die Datenausgabe im engeren Sinne ist die direkte Ausgabe in visueller Form und zwar

- ☞ auf dem Bildschirm,
- ☞ auf Papier über Drucker oder Plotter oder
- ☞ auf Mikrofilm (COM = Computer Output on Microfilm)

3.2.2.3. Akustische Datenausgabe

Bei der aktustischen Datenausgabe (Sprachausgabe) werden zwei Verfahren unterschieden

- ☞ Halbsynthese: Als Sprachmuster eingegebene Wörter oder Wortfolgen werden digital gespeichert und zur Ausgabe wieder in analoge Schwingungen umgewandelt (Sprachwiedergabesysteme).
- ☞ Vollsynthese: Digital gespeicherter Text wird anhand von Sprachlauten, die als sogenannte Phoneme (Sprachlaute) oder Diphone (Lautübergänge) gespeichert sind, in Sprachsignale umgewandelt (Sprachsynthesesysteme).

In Betracht kommen für die Datenausgabe

- ☞ der Versand von DV-Ausdrucken (Drucklisten),
- ☞ das Verschicken von Fernkopien der Drucklisten über Telefax,
- ☞ der Versand von Datenträgern (Magnetband, Diskette, etc.),
- ☞ der Versand von Microfiches,
- ☞ die Datenübertragung über Netze mit Druck beim Empfänger oder
- ☞ die Ausstattung der Empfänger mit Bildschirmterminals oder Mikrocomputern, mit denen der Empfänger über Fest und Funknetze auf die Daten zugreifen kann.

Zusammen mit der Dateneingabe helfen folgende Fragen, die richtige Form der Datenausgabe zu ermitteln:

- ☞ Wo und an wie vielen Stellen werden die Daten benötigt?
- ☞ Wie aktuell müssen die Daten sein?
- ☞ Wie viele Daten sind zu welchen Zeiten dafür zu übermitteln?
- ☞ Wie oft verändern sich die Daten?

3.3. Vernetzung

Ein *Rechnernetz* (Rechnerverbundsystem) ist ein Verbund mehrerer getrennt arbeitender, selbständiger Rechner, die durch einen Übertragungsweg (Leitung, Funk etc.) miteinander kommunizieren können.

Weiter wird unterschieden zwischen einem *lokalen Rechnernetz*, LAN (local area network), und einem *Rechnerfernnetze* (WAN, wide area network)[52].

Bei einem Rechnerfernnetz, WAN, handelt es sich um Netze, in denen geographisch „weit entfernte" Rechner miteinander verbunden sind. Ein LAN dagegen besteht nach derzeitiger Rechtslage aus einem Netzverbund von Rechnern, die paarweise nicht weiter als 25 km voneinander entfernt sind. Relevant ist die Tatsache, dass die Besitzer des Netzes (also i. d. R. das Unternehmen) die Leitungen selbst verlegen und betreiben dürfen (im Gegensatz zu den öffentlichen Telekom-Netzen, die bis Ende 1997 dem Monopol unterlagen).

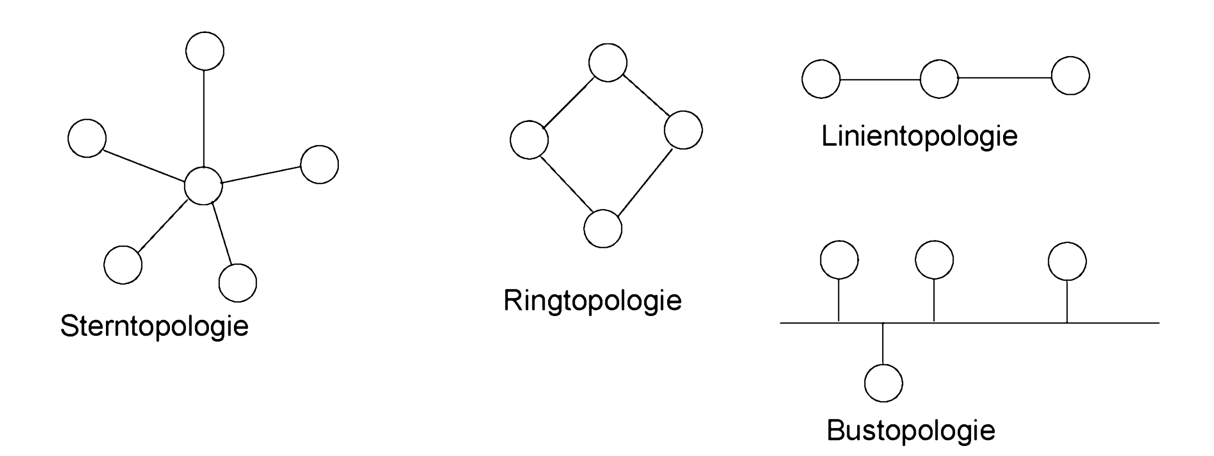
In der sogenannten *Verbundart* unterscheidet man den Zweck der Rechnervernetzung:

1. Last- oder Kapazitätsverbund
   Bei Aus- und Belastungsschwankungen hilft ein Teil der Netzrechner einem anderen aus.
2. Geräte- oder Betriebsmittelverbund
   Es werden Peripheriegeräte von mehreren Rechnern gemeinsam genutzt, z.B. Drucker, die nicht an jedem Rechner einzeln angeschlossen werden müssen (Prinzip der Printserver oder bei Peer-to-Peer-Verbindung).
3. Funktionsverbund
   Nutzung von Programm(funktion)en, die auf einem anderen Computer verfügbar sind (Application Server).
4. Datenverbund
   Verteilte oder entfernt gelagerte Datenbestände werden (gemeinsam) genutzt (File Server, Database Server).
5. Nachrichten- oder Kommunikationsverbund
   Informationen werden zwischen den Benutzern der einzelnen vernetzten Rechner ausgetauscht (Mail und News Server, Groupware (z. B. Lotus Notes)).

Rechnernetze können graphisch durch Knoten- und Verbindungsstrecken visualisiert werden. Die einzelnen Knoten entsprechen dabei den Rechnern, deren geographische Anordnung wird als *Netzwerktopologie* bezeichnet. Die im wesentlichen auftretenden Formen sind nachstehend skizziert.

---

[52] Daneben kursieren auch Begriffe wie *MAN* - metropolitan area network.



Bei der *Sterntopologie* ist jeder Teilnehmer mit einem bestimmten Rechner, der Zentrale, verbunden (wie bei einer Telefonanlage). Einsparungen von Leitungen und die leichte Erweiterbarkeit stehen der Abhängigkeit von der Zentrale (Überlast, Störungen, Ausfall) gegenüber.

Bei der *Ringtopologie* sind alle Rechner gleichberechtigt, es gibt keine Zentrale. Auch bei großer Teilnehmerzahl ist der Leitungsaufwand gering; diese Struktur ist allerdings anfällig gegen hohe Belastungen und fällt (in der Regel ganz) aus, wenn ein Rechner nicht (mehr) läuft.

Die *Linientopologie* ist ein Spezialfall der Ringstruktur, ein sogenannter *offener Ring*.

Bei der sehr häufig praktizierten *Bustopologie* sind die Teilnehmer über einen gemeinsamen Strang, den sogenannten *Bus*, miteinander verbunden. Auch bei Ausfall eines (beliebigen) Rechners können die anderen weiter miteinander kommunizieren.

Eine übliche logische Einteilung der verschiedenen Schichten, die beim Netzwerkverbund zum Tragen kommen, wird durch das ISO Referenzmodell (Schichtenmodell)[53] gegeben.

7	Anwendungsschicht	application layer	Definition erlaubter Anwendungen: Datenbankabfrage, Prüfung von Zugangsberechtigungen, Buchung, allgemeine Rechnernutzung
6	Darstellungsschicht	presentation layer	Festlegung der Bedeutung ausgetauschter Daten: Codierungen, Sprache, Graphik
5	Kommunikations-steuerungsschicht	session layer	Festlegung der Kommunikation zwischen den Teilnehmern in Form von Sitzungen (sessions): Steuerung des gesamten Benutzerdialogs vom Login bis zum Logout
4	Transportschicht	transport layer	Steuerung und Überwachung der logischen Verbindung zwischen Sender und Empfänger

[53] seit 1991 als DIN ISO 7498 genormt

3	Vermittlungsschicht	network layer	Auf- und Abbau des gesamten physischen Übertragungsweges - Anwahl und Bestätigung des Verbindungsaufbaus usw.
2	Sicherungsschicht	link layer	Sicherung der Schicht 1 - fehlergesicherte Übertragung (Fehlererkennung und Fehlerkorrektur)
1	physikalische Schicht	physical layer	Ungesicherte Übertragung von Bitfolgen über eine Übertragungsstrecke; Vereinbarungen über Schnittstelle, die Übertragungsrate sowie die angewendeten Übertragungsverfahren

3.4. Peripherie

Unter *Peripherie* versteht man Geräte zur Ein- und Ausgabe von unterschiedlichsten Daten wie etwa Tastatur, Bildschirm, Maus, Drucker, Scanner etc. aber auch Speichermedien wie Disketten-, CD-ROM- und Festplattenlaufwerke.

3.4.1. Eingabegeräte

1. Tastatur
   Die Tastatur ist zusammen mit dem Monitor (Bildschirm) die Standardschnittstelle zwischen Mensch und Rechner. Sie ähnelt mit einigen Abweichungen der guten alten Schreibmaschinentastatur, die Belegung der einzelnen Tasten wird jedoch über das Betriebssystem gesteuert und kann auf diese Weise nationalen Zeichensätzen oder bestimmten Anforderungen angepasst werden. Jede Taste liefert einen eigenen sogenannten Scan-Code, der auch Tastenkombinationen erkennen lässt. Die meisten Tasten können daher in zwei- oder dreifachen Kombinationen verwendet werden (mit den jeweiligen Sondertasten). So schließt zum Beispiel die Tastenkombination [Alt]+[F4] unter Windows üblicherweise ein Programm.
2. Maus
   Die Maus dient zur Manipulation von graphikorientierten Oberflächen und Texten. Man kann mit ihr einerseits die Cursorposition am Bildschirm verändern, andererseits durch das sogenannte "Anklicken" bestimmte Programmaktivitäten hervorrufen. Für viele Softwarepakete zählt die Maus zur Hardware-Voraussetzung, bzw. wird eine Arbeit ohne sie außerordentlich mühsam.
   Um arbeiten zu können, benötigt die Maus spezielle Programme (zusätzlich zu den Anwenderprogrammen), die Treiber genannt werden. Bei diesen ist besonders auf Kompatibilitätsprobleme zu achten.
3. Trackball
   Der Trackball kommt besonders bei Notebook- und Laptop-Computern zum Einsatz und manipuliert - ähnlich wie die Maus - die aktuelle Cursorposition.
4. Scanner
   Scanner sind Geräte, die es ermöglichen, Vorlagen unterschiedlichster Art optisch abzutasten und als graphische Information am Computer zu verspeichern. Man unterscheidet insbesondere zwischen Handscannern und Flachbettscannern.

Scanner sind in der Lage, je nach Hersteller und Preis, Schwarzweiß- oder Farbvorlagen in nahezu beliebiger Auflösung zu digitalisieren. Als Ergebnis eines solchen Abtastvorganges entsteht eine Graphikdatei, die unterschiedliche Formate aufweisen kann und - in Abhängigkeit von der Auflösung der Vorlage und des verwendeten Dateiformates - von erheblicher Größe sein kann. Das Standardformat beim Scannen ist das Tagged Image File Format (TIFF[54]).

3.4.2. Ausgabegeräte

1. Monitor (Bildschirm)
   Der Monitor dient zur Ausgabe von Texten oder Graphiken und benötigt zu deren Darstellung eine Bildschirmkarte, die in die Hauptplatine des Computers eingesteckt ist. Man unterscheidet bei der Darstellung zwischen Text- und Graphikmodus.
   Entscheidend für die Qualität einer Bildschirmdarstellung sind die verwendete Auflösung in Pixeln (Bildpunkten), die Bildschirmgröße und die Farbtiefe; daneben spielen beim Monitor der sogenannte Lochabstand und die Strahlung eine Rolle. Verbindungsglied zwischen dem eigentlichen Rechner und dem Bildschirm ist die Graphikkarte.
   Die heute üblichen Bildschirme besitzen eine Bildschirmdiagonale von 17 Zoll und können eine Auflösung von mindestens 640 x 480 dpi (dots per inch) darstellen bei einer Lochgröße von ca. 0,28 mm.
   Im Bereich der Graphikkarten existieren sehr viele Anbieter mit unterschiedlichsten Standards. Im Bereich der PCs sind bzw. waren u.a. gebräuchlich der CGA-Standard (mit stolzen 16 Farben), der VGA-Standard (mit den o.e. 640x480-Auflösung und einer Farbtiefe von 256 Farben) sowie der SVGA-Industriestandard mit 1024x768 Punkten und ca. 16 Millionen Farben.
   Daneben gibt es noch sehr leistungsfähige Graphikkarten mit Auflösungen über 1240x1024 dpi, die auch auf Workstations zu Anwendung kommen.

2. Drucker
   Die Situation am Markt ist bei Druckern noch unübersichtlicher als bei der PCs insgesamt. Für jede Anwendung und nahezu jede Preisklasse gibt es ein unüberschaubares Angebot von Druckern unterschiedlichster Qualität. Grundsätzlich kann bei modernen Druckern unterschieden werden (u.a.) zwischen Nadeldruckern (z.B. heute noch für Belege mit Durchschlägen erforderlich), Tintenstrahldruckern und Laserdruckern. Zur Bewertung eines Druckers sind u.a. die Druckgeschwindigkeit, das Schriftbild oder die Graphikqualität sowie ggf. die Farbdarstellung heranzuziehen.

3.4.3. Speichermedien

1. Disketten und Diskettenlaufwerke
   Die leicht transportablen Disketten werden auch Floppy Disks genannt. Sie sind dünne, aus Magnetfolie produzierte Scheiben in unterschiedlichen Hüllen und in verschiedener Größe. Je nach Größe der jeweiligen Diskette und der Dichte ihrer Spuren können bestimmte Datenmengen gespeichert werden.
   Heute sind im wesentlichen noch zwei Diskettentypen mit dem Format 3,5 Zoll in

[54] Die übliche Dateinamenerweiterung unter DOS/Windows hierfür ist .TIF, da längere Erweiterungen erst mit Windows NT bzw. 95 möglich geworden sind.

Verwendung: die einfacheren mit einer Kapazität von 720 Kilobyte und die „high density"-Disketten mit einem Fassungsvermögen von 1,44 Megabyte.
Disketten werden vor allem zum Transfer kleinerer Datenmengen, als Installationsmedium für Anwenderprogramme und in geringerem Umfang auch noch zu Sicherungszwecken (vor allem im Hausgebrauch) verwendet.
Disketten haben eine begrenzte Lebensdauer; sie sollten vorsichtig behandelt und nicht auf einer Heizung oder neben dem Fernseher aufbewahrt werden.

2. Festplatten
Festplatten oder englisch Harddisks gestatten das Speichern umfangreicher Programme und Daten. Festplatten sind prinzipiell abgeschlossene Miniatur-Magnetplattenlaufwerke, also eine Einheit aus Laufwerk und Speichermedium.
Die Speicherkapazitäten beginnen heute bei einigen hundert MByte und reichen bis in den GByte-Bereich, wobei die Faustregel gilt, dass mit zunehmender Größe der Festplatte die Zugriffszeit abnimmt. Marktübliche Zugriffszeiten liegen heute zwischen ca. 8 und 20 Millisekunden.
Festplatten bestehen aus einem Stapel einzelner, in der Regel beidseitig beschichteter Aluminiumscheiben, zwischen denen sich Schreib- und Leseköpfe befinden. Da der Abstand zwischen Speichermedien und den Köpfen extrem knapp ist, müssen Festplatten von Erschütterungen und Verunreinigungen geschützt werden. Wie auch bei der Floppy Disk benötigt eine Festplatte einen Harddisk-Controller, wobei auch hier aufgrund der unterschiedlichsten Standards auf die Kompatibilität zu achten ist.
Die wichtigsten Typen von Festplatten im PC-Bereich sind IDE-Platten und SCSI-Platten.

3. USB-Sticks
An dem seit einigen Jahren üblichen USB-Anschluss (*universal serial bus*) hat sich ein etwa feuerzeuggroßes Gerät etabliert, der sogenannte *USB-Stick*. Dabei handelt es sich um einen permanenten Datenspeicher, der über die USB-Schnittstelle mit dem Computer kommuniziert. USB-Sticks gibt es in verschiedenen Speichergrößen - von 16 MB bis über 1 Gigabyte. Einige benötigen eine separate Stromversorgung, andere können über den USB-Anschluss mit Strom versorgt werden.

4. Weitere Speichermedien
Neben Disketten und Festplatten sind heutzutage CD-ROMs (compact disc, read only memory) üblich; dabei handelt es sich (wie bei Musik-CDs) um (für den Normalanwender) nur lesbare[55] Medien, die insbesondere zur Auslieferung von Software und im Multimedia-Bereich eingesetzt werden.
Sogenannte Zip-Laufwerke sind ein weiterer Versuch, diskettenähnliche Medien zum Einsatz zu bringen; allerdings sind derzeit die verwendeten Formate bei Zip-Laufwerken stark herstellerabhängig, was den Austausch solcher Medien erschwert.

[55] Es sei nur am Rande erwähnt, dass es seit geraumer Zeit auch sogenannte „CD-Brenner" gibt, mit denen CD-„Rohlinge" einmal beschrieben werden können. Wie fast alle Technologien so ist auch diese in den vergangenen Jahren preislich für Otto Normalverbraucher immer erschwinglicher geworden.

3.4.4. Kommunikationsschnittstellen

1. Serielle Schnittstellen
   Die serielle Schnittstelle (auch V.24, RS232 oder RS422-Schnittstelle genannt) erlaubt die serielle Datenübertragung im synchronen oder asynchronen Modus. Die einzelnen Bits (eines Bytes) werden dabei hintereinander auf einer Leitung übertragen. Für diese Übertragung müssen beide Geräte (Sender und Empfänger) auf gleiche Weise eingestellt sein. parametrisiert sein.
   PCs verfügen häufig über zwei serielle Schnittstellen, die dann als COM1 und COM2 bezeichnet werden. Gegebenenfalls werden weitere serielle Schnittstellen dann mit COM3, COM4 usw. bezeichnet.

2. Parallele Schnittstellen
   Die parallele oder sogenannte „Centronics"-Schnittstelle wird fast ausschließlich für den Betrieb von Druckern verwendet. Dabei werden bis zu 8 Bit (eines Bytes) parallel auf 8 Leitungen übertragen, was im Idealfall eine achtfache Geschwindigkeit der seriellen Übertragung bedeutet. Hierbei können jedoch nur kurze Distanzen (zwei bis maximal fünf Meter) überbrückt werden. Bei Personal Computern (unter MS-DOS, OS/2 und Windows) heißen diese Schnittstellen LPT1, LPT2 usw., wobei in der Regel heute nur noch eine parallele Schnittstelle LPT1 vorhanden ist, die auch mit PRN bezeichnet wird.

4. ALGORITHMEN UND DATENSTRUKTUREN

In diesem Kapitel wollen wir uns mit Grundlagen beschäftigen, die für die Software-Entwicklung, zum Teil aber auch für das Erstellen komplexer Daten- und Beziehungs- und Objektmodelle von großer Bedeutung sind. Eine der ältesten und gängigsten Literaturempfehlungen hierzu ist das Buch von [Wirth].

Wir beginnen mit Datenstrukturen, die heutzutage von Standard-Programmiersprachen bereits zur Verfügung gestellt werden[56].

4.1. Grundlegende Datenstrukturen

Werden in einer höheren Programmiersprache[57] Variablen (Speicherplätze) angelegt, dann ist es üblich, diesen einen sogenannten *Datentyp* zuzuordnen. Darunter versteht man die Festlegung einer Wertemenge, aus der die Variable zu jedem Zeitpunkt genau einen Wert annehmen oder besitzen kann. Welchen Speicherplatz, d.h. welche Anzahl von Bits, eine Variable eines gewissen Datentyps benötigt, dies hängt im wesentlichen von der Kardinalität[58] der zugrunde liegenden Wertemenge ab.

4.1.1. Einfache Datentypen

Betrachten wir die Programmiersprache (ANSI-)C. Dort finden wir (u.a.) die vordefinierten einfachen Datentypen `char`, `int` und `float`. Dabei handelt es sich um Typen, mit denen jeweils ein einzelnes Zeichen[59] (des ASCII-Codes), eine ganze Zahl bzw. eine Gleitkommazahl abgespeichert werden können (vgl. hierzu Abschnitt 1.5.1 auf S. 33 zur Codierung von Zahlen).

In jeder Programmiersprache gelten Regeln, die es erlauben festzustellen, von welchem Datentyp ein bestimmter Ausdruck, eine Konstante oder eine Variable ist. Bei jeder Operation sind ein oder mehrere Operanden bestimmter Datentypen beteiligt, und die jeweilige Programmiersprache legt fest, welche Datentypen mit welchen verträglich sind und wie der Ergebnisdatentyp aussieht.

---

[56] Naturgemäß stellt nicht jede Programmiersprache alle Datenstrukturen bereit; hier ist es dann besonders wichtig, dass (angehende) Wirtschaftsinformatiker/innen in der Lage sind, diese Datenstrukturen eigenständig zu modellieren! So verfügt beispielsweise C nicht über einen eigenen Datentyp zur Implementation von Mengen.

[57] Wir werden nachfolgend die Sprachen C und Pascal für Beispiele heranziehen; in den meisten Fällen ist die konkrete Programmiersprache jedoch nicht entscheidend.

[58] Die Kardinalität einer Menge ist die Anzahl der Elemente dieser Menge bzw. ∞ im Falle von unendlichen Mengen.

[59] An dieser Stelle soll nicht von Interesse sein, dass in C eine `char`-Variable intern rein numerisch gespeichert wird, und dass mit ihr auch ebenso wie mit einem `int` gerechnet werden kann.

4.1.1.1. Numerische Datentypen: Ganze und Gleitkommazahlen

Hierzu wieder ein Beispiel: bleiben wir in C, dann erläutert der nachfolgende Code-Auszug einiges zu der hier erwähnten Typenverträglichkeit.

`int i=1, j=2, k=3;`	i, j und k werden als `int`-Speicherplätze deklariert und definiert.
`float x=1, y=2;`	x und y entsprechend als `float`.
`k = i + j;`	Die Addition zweier `int`-Werte ergibt naturgemäß wieder einen Wert vom Typ `int`.
`x = k;`	Und eine Ganzzahl kann an eine Gleitkomma-Variable zugewiesen werden, hierbei wird die interne Darstellung automatisch konvertiert.
`x = i / j;`	Hier findet zunächst die Division von `i` mit `j` statt. Da beides `int`-Variablen sind, handelt es sich hier (bei der Programmiersprache C) um eine Ganzzahldivision „modulo", d.h. das Ergebnis ist hier 0. Dieser (Ganzzahl-)Wert 0 wird dann an die `float`-Variable zugewiesen, wobei wiederum automatisch in die Gleitkommadarstellung konvertiert wird. `x` hat aber dann den Wert 0 - nicht 0.5 !
`i = 3.1415926;`	Die rechte Seite dieser Zuweisung ist eine Gleitkommazahl; links von der Zuweisung steht die Angabe einer Ganzzahl-Variablen. In manchen Programmiersprachen, beispielsweise Pascal, ist dies ein Fehler! In anderen Sprachen (wie auch C) wird diese Typenprüfung nicht so streng gesehen, vor der Zuweisung wird aber der Wert 3.1415926 abgeschnitten („trunkiert", „truncated"), `i` erhält „nur" den Wert 3.

Bereits in dieser Aufstellung sind zwei einfache Datentypen aufgetreten. Einmal der in C `int` genannte Typ für die Darstellung und Verarbeitung ganzer Zahlen. Und dann als zweites ein Datentyp zur Abspeicherung von (Gleit-)Kommazahlen (beispielsweise 1,23 - wobei es in der Programmierung üblich ist, hierfür 1.23 zu schreiben, also den Punkt statt des Kommas als Dezimaltrennzeichen zu verwenden). In C werden Gleitkommazahlen durch die Datentypen `float` und `double` repräsentiert[60]. Hierzu und für das weitere sei auch auf Abschnitt 1.5.1 auf Seite 33 hingewiesen.

4.1.1.2. Zeichen und Zeichenketten (Strings)

Auch wenn im Digitalrechner letztlich alle Daten als Sequenzen von „0" und „1" abgelegt sind, so handelt es sich inhaltlich doch sehr häufig um nicht-numerische Informationen. Wichtigstes Beispiel sind Zeichen und Zeichenketten - beispielsweise 'a' oder "Wirtschaftsinformatik". Ein einzelnes Zeichen wird üblicherweise durch einen vereinbarten

---

[60] Der Unterschied dieser beiden Datentypen besteht in der zur Verfügung gestellten Speicherbreite, auf die wir in diesem Rahmen jedoch nicht weiter eingehen wollen. Ebenso bietet ANSI-C neben dem hier erwähnten Ganzzahltyp `int` noch einen weiteren namens `long` oder `long int`; aber auch dies wollen wir an dieser Stelle nicht weiter behandeln.

Code numerisch dargestellt[61]. Eine Zeichenkette wird dann im wesentlichen durch die Sequenz der einzelnen Zeichen repräsentiert[62].

Position	1	2	3	4	5	6	7	8	9	10	11
Zeichen(kette)	**W**	**i**	**l**	**l**	**k**	**o**	**m**	**m**	**e**	**n**	**!**
gespeicherte Zahl	87	105	108	108	107	111	109	109	101	110	33

Die Skizze illustriert dies beispielhaft auf der Grundlage der Codierung gemäß ASCII. So wird das Zeichen 'W' durch die (dezimale) Zahl 87 (hexadezimal: 0x57 = 5 * 16 + 7) dargestellt, ebenso entspricht dem 'i' die Codierung durch die Zahl 105.

Das bedeutet aber im Umkehrschluss: sieht man in einen Speicherplatz hinein, so findet man dort immer Bitfolgen, also Sequenzen aus "0" und "1"; *wie* diese dann zu interpretieren sind, wird ausschließlich über die Datentypen geregelt! Steht an einer Speicherstelle die Bitfolge, die der hexadezimalen 57 entspricht, dann kann dies dezimal 87 bedeuten, nämlich dann, wenn es sich um einen Speicherplatz handelt, der einen ganzzahligen Datentyp repräsentieren soll. Die hexadezimale 57 kann aber ebenso für das Zeichen 'W' stehen, sofern der Datentyp angibt, dass die betreffende Information gemäß ASCII codiert sein soll.

4.1.1.3. Aufzählungstypen (Enumerations)

Etwas weniger nutzen Software-Entwickler die Möglichkeit zahlreicher Programmiersprachen, eigene sogenannte Aufzählungstypen zu deklarieren, mit denen der Code wesentlich selbstsprechender gestaltet werden kann.

Hierzu ein konkretes Beispiel (wiederum in C[63]). Nehmen wir an, wir wollten etwas für die sieben Tage der Woche codieren. Dann lägen folgende Fragmente recht nahe.

```
int wochentag=0; /* 0 steht für Montag */

for (wochentag=0; wochentag<=6; wochentag++)
{
   /* hier nun der Code fuer den betreffenden Wochentag */
}

if (wochentag==6)
{
   printf("Heute ist Sonntag!")
}
```

[61] Wie bereits erwähnt: im ASCII wird das Zeichen 'A' repräsentiert durch die Dezimalzahl 65.

[62] „Im wesentlichen" soll bedeuten, dass es - je nach Programmiersprache - zu einer Zeichenkette entweder noch die Information geben kann, wie lang sie ist, oder aber (wie in C) mit einem speziellen Abschlusszeichen das Ende markiert wird.

[63] Auch Pascal bietet Aufzählungstypen an; hier kontrolliert der Compiler sogar noch sehr viel stärker als in C, dass die Werte des Datentyps eingehalten werden.

Sehr viel lesbarer wird der Code dagegen bei Verwendung eines Aufzählungstyps[64]:

```
enum wochentagstyp
{
   montag, dienstag, mittwoch, donnerstag, freitag, samstag, sonntag
};
enum wochentagstyp wochentag;

for (wochentag=montag; wochentag<=sonntag; wochentag++)
{
   /* hier nun der Code fuer den betreffenden Wochentag */
}

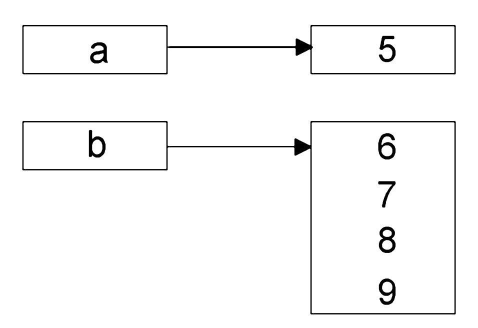
if (wochentag==sonntag)
{
   printf("Heute ist Sonntag!");
}
```

Je nach Programmiersprache und Compiler-Einstellungen überwacht das Entwicklungssystem hierbei auch darüber, dass keine ungültigen Werte verwendet werden. Auch dies ist ggf. ein Pluspunkt gegenüber der zuerst gezeigten primitiven Ganzzahl-Variante.

4.1.1.4. Pointer (Zeiger)

Schließlich muss noch ein im Kern einfacher Datentyp vorgestellt werden, der aber vielen Anfänger(inne)n der Programmierung alles andere als einfach fällt: der Pointer- oder Zeigertyp.

Die nachfolgende Skizze illustriert den grundlegenden Sachverhalt.



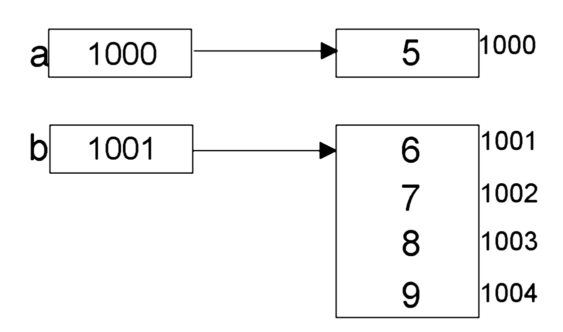
Die hier dargestellten Variablen (Speicherplätze) a und b beinhalten nicht selbst „interessante" Werte, sie dienen vielmehr dazu, auf andere Speicherplätze zu verweisen (zu „zeigen"). In technischer Hinsicht beinhalten Variablen eines Pointertyps also Speicheradressen.

Erweitern wir das obige Bild, dann können wir (fiktive) Adressen für die rechts dargestellten Speicherplätze angeben, hier die Nummern 1000 bis 1004. Die übliche Sprechweise ist: der Pointer a zeigt auf den Speicherplatz 1000, in dem die Zahl „5" gespeichert ist. Damit wird

---

[64] Technisch wandelt der C-Compiler die hier festgelegten Namen um in die Zahlen 0 bis 6; es bleibt also intern alles beim alten. Allerdings ist m.E. der von Menschen zu lesende Quellcode sehr viel verständlicher geworden. Insbesondere muss nicht kommentiert oder darüber nachgedacht werden, ob nun die Zahl 0 für den Montag oder für den Sonntag steht.

nichts anderes ausgesagt, als dass in der Pointervariablen a die Adresse 1000 gespeichert steht; zusammen mit dem durch den Datentyp festgelegten Wissen, dass diese 1000 eine Adresse im Speicher angibt, wird die obige Sprechweise deutlich.



4.1.2. Strukturierte Datentypen

Unter einem *strukturierten Datentyp* versteht man einen Datentyp, bei dem sich die anzunehmenden Werte selbst zusammensetzen aus kleineren Einheiten. Im Gegensatz oder Vergleich dazu sind die elementaren Datentypen eher *atomar* zu nennen.

4.1.2.1. Array (Feld)

Das bekannteste Beispiel eines strukturierten Datentyps ist der *Array-* oder *Feldtyp*. Hierbei handelt es sich um eine Zusammenfassung von endlich vielen Komponenten desselben Grundtyps.

Beispiel: Für eine Studiengruppe von zwanzig Personen seien die (ganzzahligen) Punktezahlen der Informatik-Klausur festzuhalten. Sind die zwanzig Personen der Reihe nach angeordnet (oder durchnummeriert), so genügt die nachstehend bildhaft dargestellte Information.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
80	75	94	28	55	69	75	79	84	99	93	12	45	39	70	76	74	56	84	91

Eingerahmt sind zwanzig Kästchen dargestellt, in denen jeweils eine ganze Zahl Platz findet. Darüber steht der jeweilige Index. Das Ganze wird Array oder Feld genannt, die Deklaration sieht in Pascal wie folgt aus.

```
type arraytyp = array[1..20] of integer;
```

Damit ist zunächst nur der Datentyp deklariert worden, d.h. die „Absichtserklärung", dass künftig mit dem Namen `arraytyp` genau solche Gebilde verwaltet werden sollen.

Einen konkreten Speicherplatz a mit dieser Struktur erhält man in Pascal dann wie folgt.

```
var a : arraytyp;

a[1] := 80;
a[20] := 91;
```

Auch in C gibt es (natürlich) Arrays, allerdings beginnt dort die Indizierung immer bei 0. Das obige Bild müsste also modifiziert werden, damit der nachfolgend gezeigte Code in ANSI-C korrekt veranschaulicht wird.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
80	75	94	28	55	69	75	79	84	99	93	12	45	39	70	76	74	56	84	91

Hierzu passt dann der folgende C-Code-Auszug.

```
typedef int arraytyp[20];

arraytyp a;

a[0] = 80;
a[19] = 91;
```

4.1.2.2. Record (Struct)

Während bei einem Array stets gleichartige Komponenten zusammengefügt werden, die dann über einen in der Regel numerischen Index angesprochen werden können, bietet ein Record (oder eine Struktur, ein struct, ein Verbund, - lauter synonyme Begriffe) die Möglichkeit, eine Einheit aus verschiedenartigen Komponenten zu bilden. Schon aus technischen Gründen sind diese Komponenten dann aber nicht mehr über einen Index sondern über einen jeweiligen Komponentennamen verfügbar. Solche Records treten beispielsweise bei Datenbanken auf, etwa wenn für einen Kunden dessen Nachname, Vorname und die Kundennummer verwaltet werden.

In C kann dies dann so aussehen.

```
/* Vereinbarung des Datentyps */
struct kundentyp
{
   char nachname[80];
   char vorname[80];
   int kundennummer;
};

/* Deklaration und Definition der Variablen */
struct kundentyp kunde1, kunde2;

kunde1.kundennummer = 123;/* Zuweisung der Kundennummer fuer kunde1   */
puts(kunde2.nachname);    /* Ausgabe des Nachnamen auf den Bildschirm */
```

kunde1

nachname	vorname	kundennummer

4.1.2.3. Menge (Set)

Außer Arrays und Records ist ein dritter, häufig eingesetzter strukturierter Datentyp die *Menge*, englisch: *set*. Wie beim Array werden auch bei einer Menge gleichartige Dinge zusammengefasst. Während beim Array jedoch ein konkreter Wert auch mehrfach auftreten kann, ist bei einer Menge mathematisch nur relevant, ob ein Element überhaupt vorkommt oder nicht. Auch die beim Array mit verwaltete Reihenfolge spielt hier keine Rolle. Die Mengen A = { 1, 2, 3 } und B = { 2, 3, 1, 3 } sind also identisch.

Die Programmiersprache C bietet keinen fertigen Datentyp für Mengen an, Pascal immerhin Mengen von einfachen Datentypen (mit einer je nach Compilersystem begrenzten Größenbegrenzung). Sehen wir uns also ein einfaches Beispiel einer Menge über der Grundmenge { 1, 2, 3, 4, 5 } an.

```
type mengentyp = set of 1..5;  { Mengen über der Grundmenge 1 .. 5 }
var menge1 : mengentyp;

menge1 := [ ];{ Zuweisung der leeren Menge, also der Menge, die kein Element beinhaltet }
menge1 := menge1 + [ 1 ];  { Aufnahme der Zahl 1 in die Menge }
```

Werfen wir kurz einen Blick auf den Speicherbedarf einer Menge. Es mag auf den ersten Blick umständlicher und komplexer aussehen, als es wirklich ist. Wir benötigen in der Tat nur für jedes Element der Grundmenge (hier im Beispiel also {1,2,3,4,5}) ein Bit: dieses wird auf 0 gesetzt, wenn das betreffende Element nicht in der fraglichen Menge ist, andernfalls auf 1.

Die Menge { 1, 3, 4 } wäre also durch die Bitfolge 10110 darzustellen.

Ist	1	2	3	4	5	in der Menge?
1=ja:	1	0	1	1	0	

Mengen sind für die Software-Praxis eine sehr große Hilfe. Oftmals ist es nicht relevant, wie oft ein Ereignis eingetreten ist, sondern nur, ob bzw. dass es eingetreten ist. In einer Applikation, die mehrere Fenster zur Bearbeitung zulässt, ist es beispielsweise wichtig, ob in einem Fenster Änderungen stattgefunden haben, damit beim Schließen des Fensters nachgefragt werden kann, ob gespeichert werden soll. Hierfür muss aber nicht protokolliert werden, wieviele Änderungen es wirklich gegeben hat.

4.1.2.4. Datei (File)

Die bislang vorgestellten Datentypen haben alle eines gemeinsam: eine konkrete Realisierung hat nur endlich viele Werte, wobei die maximale Anzahl von vorneherein festgelegt ist. Ein Array hat maximal so viele Einträge, wie es bei der Deklaration mitgegeben bekommen hat. Ein Record hat nur die endlich vielen Komponenten, die bei der Deklaration explizit angegeben wurden. Eine Menge (in der EDV) schließlich ist stets eine Menge über einer zuvor festgelegten endlichen Grundmenge[65].

---

[65] Für den Mathematiker bedeutet dies natürlich sofort: nicht alle mathematischen Mengen sind in einem (Pascal-)Programm darstellbar.

Demgegenüber ist eine *Datei* (englisch: ein *file*) vom Datentypkonzept her eine „beliebig lange" (faktisch schließlich wieder endliche) Sequenz von Einzelwerten - meist Bytes (bzw. Zeichen)[66].

Man spricht von einer sequentiellen Datei (und von sequentieller Dateiverarbeitung), wenn man - ähnlich wie bei einem Videoband - die Informationen nur der Reihe nach von vorne nach hinten abspeichern bzw. lesen kann. Um den siebten Datensatz abzurufen, muss man also gedanklich die ersten sechs Datensätze „vorspulen".

Daneben gibt es aber auch Datei(art)en mit Direktzugriff, im Fachjargon *random access* genannt. Hier kann - ähnlich wie bei einem Array - gezielt ein einzelner Datensatz angesprochen werden.

4.2. Dynamische Datenstrukturen

Die bislang besprochenen Datentypen waren im wesentlichen statisch; einzige Ausnahme stellten in gewisser Weise Dateien dar, bei denen die Größe prinzipiell beliebig anwachsen konnte, wobei aber auch hier die Struktur (z.B. sequenzielles Verarbeiten) festgelegt wurde.

Demgegenüber sind dynamische Datenstrukturen solche, bei denen erst zur Laufzeit (*runtime*) der erforderliche Speicherplatz (im Arbeitsspeicher) allokiert[67] wird. Statisch reserviert wird im minimalen Falle lediglich eine Startadresse.

Hierzu werden die bereits erwähnten Pointer (siehe Abschnitt 4.1.1.4. auf Seite 78) eingesetzt. Damit sind Datenstrukturen möglich, die (erst) während des ablaufenden Programms angelegt werden und sich dann auch noch verändern können - sprich: dynamisch sind.

Kommen wir am besten gleich zu konkreten Beispielen.

4.2.1. Einfach verkettete lineare Listen

Unter einer *linearen Liste* versteht man eine Aufeinanderfolge von nicht notwendigerweise gleichartigen Einträgen. Beispielsweise können Personen, die zu einer Gruppe gehören, in eine solche Liste eingetragen werden. Dabei kann jederzeit auch ein Sortierkriterium, z.B. nach Alter absteigend oder alphabetisch nach Nachnamen, berücksichtigt werden. An diesem Beispiel wollen wir einige Aspekte diskutieren.

Gehen wir also davon aus, dass wir zu den Personen der Einfachheit halber nur den Nachnamen, den Vornamen und das Geburtsdatum speichern wollen. Dann sind wir inhaltlich bei einem Record-Datentyp (struct in C) angekommen; dessen Deklaration könnte in ANSI-C beispielsweise aussehen wie folgt.

[66] Der wesentliche Unterschied: ein Array[1..20] ... hat definitiv zwanzig Speicherplätze des entsprechenden Basistyps; eine Datei kann zunächst einmal unbegrenzt anwachsen. Gleichzeitig - und dies ist kein Widerspruch - ist die Datei aber zu jedem Zeitpunkt natürlich nur von endlicher Länge.

[67] Allokieren bzw. Allokation von Speicherplatz bedeutet dessen Bereitstellen.

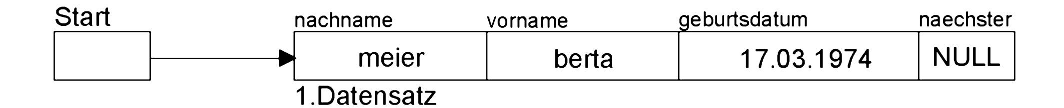
```
struct personentyp
{
   char nachname[40+1];
   char vorname[30+1];
   char geburtsdatum[10+1];  /* Darstellung: tt.mm.jjjj */
};
```

Von diesem Datentyp können nun auf statische Art und Weise einzelne Variablen oder auch ganze Arrays angelegt werden.

```
struct personentyp person1, person2;
struct personentyp personen[100];
```

Beide Varianten bedeuten aber, dass bereits zur Compile-Zeit feststeht, wieviele Personen maximal aufgenommen werden sollen bzw. können. Wir wollen aber "voll dynamisch" arbeiten können, d.h. erst zur Laufzeit des Programms soll faktisch entschieden werden, für wieviele Personen wir wirklich den Speicherplatz benötigen; wir wollen aber auch nicht durch etwa die maximal einhundert Datensätze, die wir in das obige Array `personen` maximal eintragen könnten, eingeschränkt werden.

Die nachstehende Skizze zeigt das weitere Vorgehen.



Wir nehmen uns eine Pointervariable namens `Start`, die auf den Wert `NULL` initialisiert wird, d.h. anschaulich zeigt Start zunächst auf nichts. Sobald dann die erste Person aufgenommen werden muss, wird Speicherplatz (vom Betriebssystem) angefordert und dessen Adresse `Start` zugewiesen. Da nach dieser ersten Person natürlich jederzeit noch weitere Personen aufgenommen werden können, muss es dabei wiederum eine Pointerkomponente (in unserer Skizze `naechster` genannt) geben, die zunächst ebenfalls auf `NULL` gesetzt wird.

Hierfür wird also unser `personentyp` um eine solche Komponente `naechster` ergänzt. Die betreffenden Deklarationen werden für die Beispielsprache C nachfolgend wiedergegeben[68].

```
struct personeneintrag
{
   char nachname[40+1];
   char vorname[30+1];
   char geburtsdatum[10+1];
   struct personeneintrag * naechster;
};

struct personeneintrag * Start = NULL;
/* NULL steht fuer "Zeiger auf nichts" */
```

---

[68] An dieser Stelle können naturgemäß nicht alle Probleme der Programmiersprache C behandelt werden. Es sei hier nur daran erinnert, dass Zeichenketten in C mit einem Abschlussbyte '\0' markiert werden, - daher das "+1" in den hier gezeigten char-Array-Deklarationen.

Und mit der C-Notation `datentyp * ptr;` wird `ptr` als Variable vom Typ "Zeiger auf `datentyp`" vereinbart.

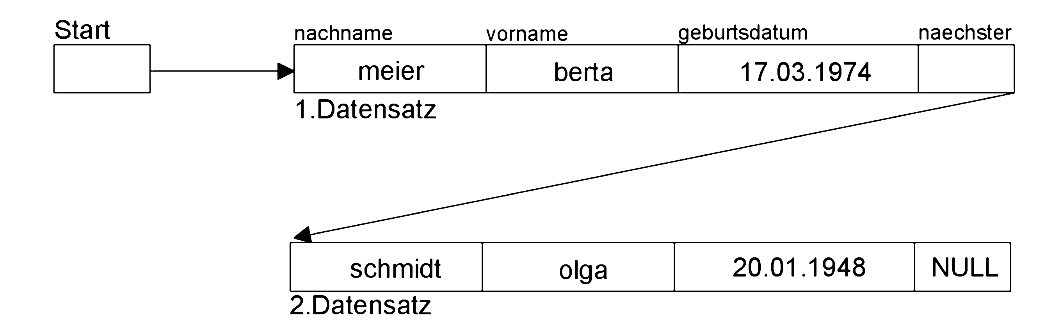
```
/* Aufnahme der ersten Person gemaess obiger Skizze */

/* Bereitstellen des Speicherplatzes mittels malloc()-Funktion69 */
Start = (struct personeneintrag *)malloc(sizeof(struct personeneintrag));

/* Zuweisen der Daten von Berta Meier */
strncpy(Start->nachname,"Meier",40);
strncpy(Start->vorname,"Berta",30);
strcpy(Start->geburtsdatum,"17.03.1974");
Start->naechster = NULL;
```

Hiermit erhalten wir die im vorher gezeigten Bild dargestellte Situation.

Nun können aber jederzeit weitere Datensätze angehängt werden; in der nachfolgenden Skizze wurde ein Datensatz für Olga Schmidt ergänzt. (Der hierzu erforderliche Code bleibe dem Leser oder der Leserin überlassen.)

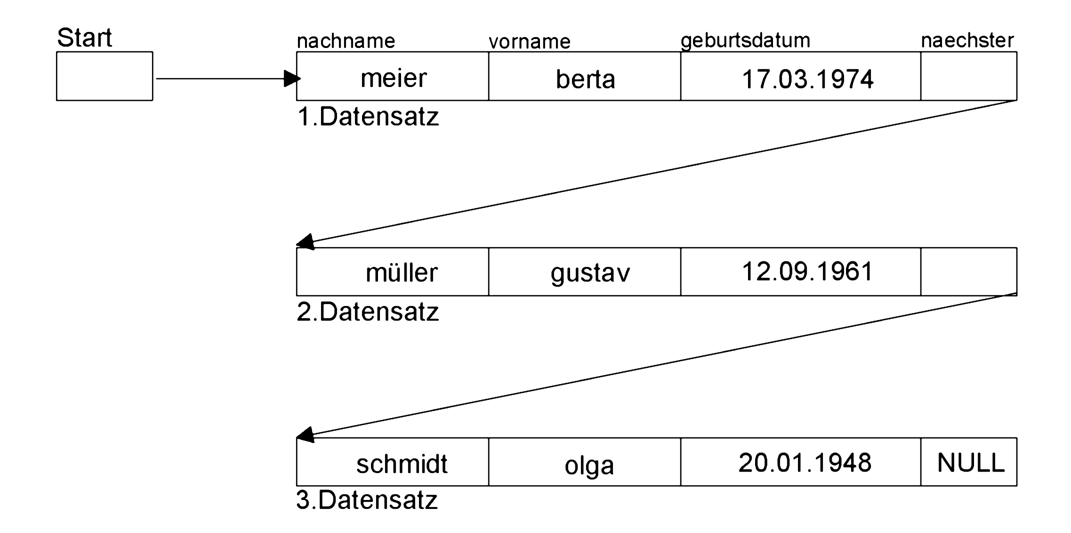


Aber lineare Listen haben noch einen weiteren positiven Aspekt: wollen wir die hier begonnene Liste nach Nachnamen alphabetisch sortiert verwalten, dann brauchen wir bei der Aufnahme eines Gustav Müller nicht physisch Daten herumzuschieben, wir fügen einfach den entsprechenden Eintrag in die Liste ein.

In der obigen Skizze heißt dies konkret: der Pointer `naechster` des 1.Datensatzes wird auf einen neuen Speicherplatz derselben Grundstruktur gesetzt, dort wird Gustav Müller eingetragen, und dessen `naechster`-Pointer wird auf den zuvor schon vorhandenen Datensatz von Olga Schmidt gesetzt.

Nach erfolgter Operation sieht die Skizze unserer linearen Liste wie folgt aus.

[69] Da es hier nur um den wesentlichen Mechanismus geht, werden hier einige für die praktische Programmierung erforderlichen Fehlerabfragen nicht abgedruckt. So könnte das Anfordern einer gewissen Menge Speicherplatz in der Praxis natürlich auch scheitern. Hierauf wollen wir an dieser Stelle jedoch nicht weiter eingehen.

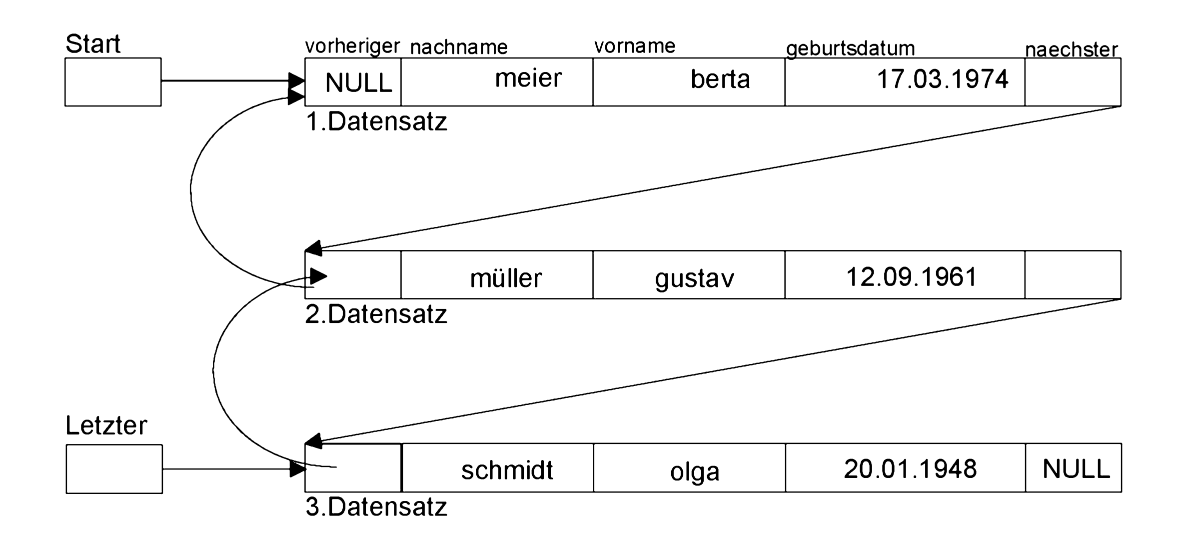


4.2.2. Doppelt verkettete Listen

Wenn Sie das obige Beispiel aufmerksam verfolgt haben, wird Ihnen aufgefallen sein, dass es ein Kinderspiel ist, die lineare Liste von Anfang bis Ende durchzuarbeiten: stets muss nur über den jeweiligen Nachfolger-Zeiger naechster zum nächsten Datensatz weitergegangen werden, solange bis dort `NULL` steht.

Es ist aber umgekehrt überhaupt nicht einfach: will man, aus welchem Grund auch immer, die Liste rückwärts durchlaufen[70], dann wird man durch die Datenstruktur überhaupt nicht unterstützt.

Für solche Fragestellungen bietet es sich an, neben dem `naechster`-Zeiger auch noch einen `vorheriger`-Zeiger zu verwalten.



Hierfür wäre auch unser Eintragstyp wie folgt anzupassen. Und es bietet sich aus Symmetriegründen an, einen Zeiger `Letzter` auf den jeweils letzten Datensatz zeigen zu

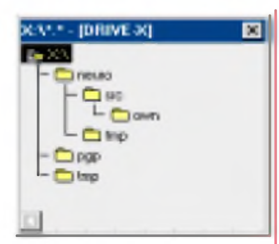
[70] Zum Beispiel möchte man die letzten drei Einträge bearbeiten...

lassen. Mittels Durchhangeln über die vorheriger-Zeiger kann nun die jetzt *doppelt verkettete Liste* ebenso bequem von hinten nach vorne abgearbeitet werden.

```
struct personeneintrag /* für doppelt verkettete Liste */
{
   struct personeneintrag * vorheriger;
   char nachname[40+1];
   char vorname[30+1];
   char geburtsdatum[10+1];
   struct personeneintrag * naechster;
};
```

4.2.3. Bäume

Wir haben lineare Listen kennengelernt als Möglichkeit, dynamischen Speicherplatzanforderungen nachzukommen. Dabei ist es Merkmal einer solchen linearen Liste, dass eine Information nach der anderen abgespeichert wird, also eine Abfolge der einzelnen Einträge vorliegt[71].



Betrachten wir jedoch das nebenstehende Bild. Dargestellt ist eine Verzeichnisstruktur, wie wir sie alle kennen. Ausgehend von einem sogenannten *Wurzelverzeichnis* (*root directory*), das in der DOS und Windows-Welt mit \ und in der Unix- und Internet-Welt mit / notiert wird, sehen wir auf der Stufe darunter (bzw. in der graphischen Darstellung eine Ebene mehr nach rechts eingerückt) die *Unterverzeichnisse* (*subdirectories*) neuro, pgp und tmp. Unterhalb von neuro (bzw. wiederum eine Ebene weiter eingerückt) finden sich dessen Unterverzeichnisse src und tmp; schließlich hat src wiederum ein Unterverzeichnis namens own.

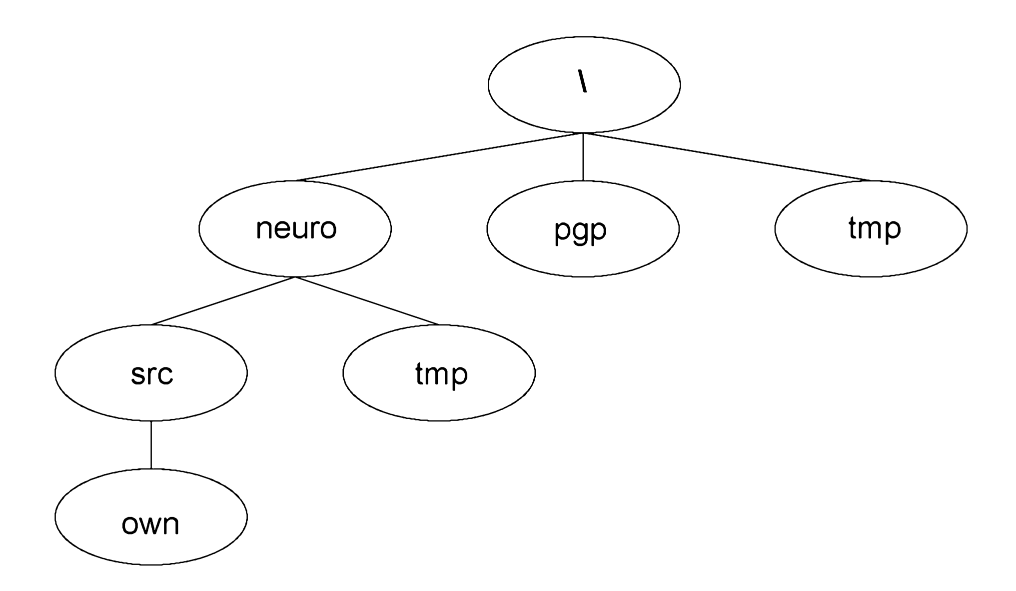
Es ist augenscheinlich, dass hierfür eine lineare Liste keine adäquate Datenstruktur wäre, denn die Verzeichnisse pgp und own sind im üblichen Sinne nicht vergleichbar. Die hier gezeigte Struktur wird in Informatik-Kreisen *Baumstruktur* genannt. Wie eine Liste ist auch ein Baum eine rekursive Struktur. Eine Liste ist entweder die leere Liste (Start=NULL), oder sie besteht aus einem ersten Knoten (Eintrag) und dem Rest, der wiederum eine Liste darstellt. Bei einer endlichen Liste ist die Anzahl der Einträge insgesamt natürlich endlich.

Entsprechend ist ein *Baum* entweder die leere Struktur oder ein Knoten, von dem aus endlich viele Verknüpfungen zu *(Teil-)Bäumen* (*subtrees*) gehen. Auch hierbei ist ein endlicher Baum eine Struktur mit nur endlich vielen Knoten.

Ähnlich wie bei der oben erwähnten Verzeichnisstruktur spricht man beim Einstiegsknoten von der sogenannten Wurzel (*root*) des Baumes. Es ist üblich, in der graphischen Darstellung eines Baumes diese Wurzel nach ganz oben oder nach ganz links zu setzen.

Das oben gezeigte kleine Beispiel sieht in einer anderen, klassischen Visualisierung aus wie folgt.

[71] Mathematisch gesprochen haben wir in einer linearen Liste eine Ordnung: zwei verschiedene Einträge x und y kommen in einer speziellen Reihenfolge vor, sind in diesem Sinne auf jeden Fall vergleichbar. Entweder kommt x vor y oder y vor x.



Hierbei wird, von oben beginnend, zunächst die Wurzel des Baumes eingezeichnet. Durch Striche verbunden wird dann in der Folgezeile die nächste Ebene dargestellt: die erste Kindgeneration der Knoten und Blätter. Ein *Knoten* ist ein Eintrag, der selbst wieder mindestens einen eigenen Folgeeintrag auf der nächsten Ebene besitzt, ein *Blatt* ist ein Eintrag, von dem aus keine weiteren Einträge vorhanden sind.

Auf der oben gezeigten ersten Ebene unterhalb der Wurzel ist `neuro` ein Knoten, `pgp` und `tmp` sind Blätter. Entsprechend ist `src` ein Knoten, `tmp` ein Blatt auf der zweiten Ebene, schließlich ist `own` das einzige Blatt auf der dritten Ebene unterhalb der Wurzel. Insgesamt besitzt dieser Baum vier Ebenen, wobei die Wurzel mit eingerechnet ist.

Besitzt ein Baum die Eigenschaft, dass jeder Knoten höchstens zwei Nachfolge-Einträge besitzt, dann nennt man ihn einen *binären Baum*. Eine lineare Liste ist, rein formal, ebenfalls ein Baum, allerdings einer, bei dem jeder Knoten nur einen Nachfolger besitzt, d.h. es handelt sich dabei um einen *entarteten Baum*.

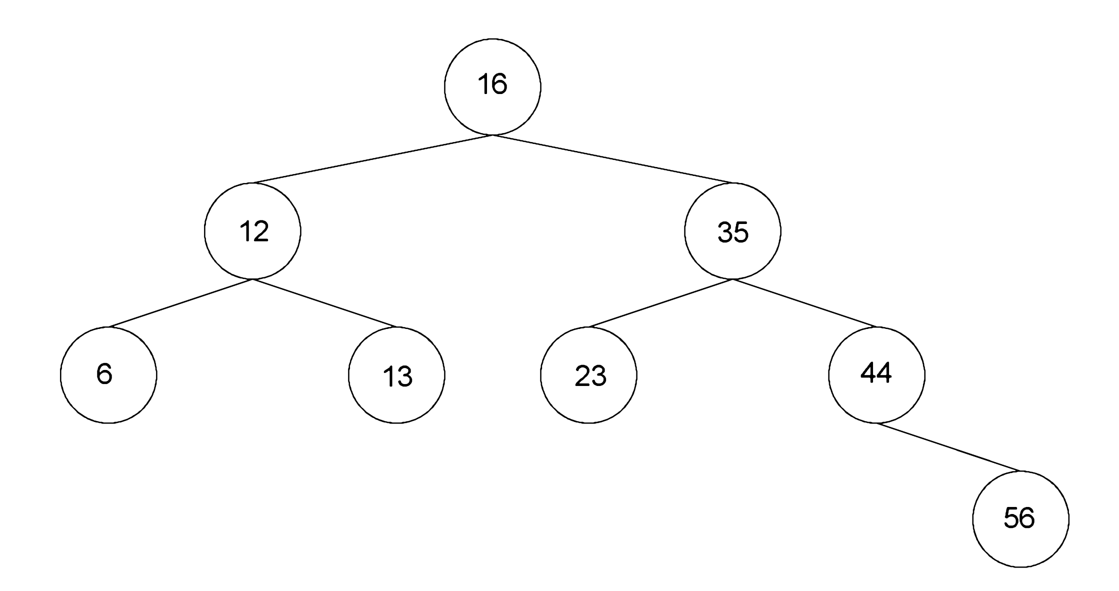
Die Implementierung einer Baumstruktur erfolgt zum Beispiel in ANSI-C selbstverständlich wieder mittels Pointerkomponenten in einem entsprechenden Struktur-Datentyp. Nachstehendes Beispiel illustriert den Knoten eines binären Baumes, wobei der eigentliche Dateninhalt auf eine Zeichenkette namens `bezeichnung` reduziert wurde.

```
struct binaerbaumknoten
{
   struct binaerbaumknoten * linker_teilbaum;
   char bezeichnung[128+1];
   struct binaerbaumknoten * rechter_teilbaum;
};
```

Neben der hier gezeigten Datentypvereinbarung wäre noch eine Variable `wurzel` für den Anfang des Baumes festzulegen.

```
struct binaerbaumknoten * wurzel;
```

Bäume können u.a. dazu verwendet werden, Daten in einer gewissen Sortierung zu verwalten und damit Suchvorgänge zu erleichtern. Nachstehend nur eine kurze Skizze, in der dargestellt wird, wie in einem binären Baum (exemplarisch) ganze Zahlen verwaltet werden können.



Beispielhaft wurden hier die Werte 6, 12, 13, 16, 23, 35, 44 und 56 abgespeichert. Dabei erfolgte die Abspeicherung so, dass für jeden Knoten gilt: im linken Teilbaum stehen nur Werte, die kleiner sind als der Wert in dem betreffenden Knoten selbst; entsprechend sind im rechten Teilbaum nur Werte zu finden, die größer sind. Die Wurzel trägt den Wert 16, in ihrem linken Teilbaum sind nur Werte kleiner als 16 zu finden: 6, 12 und 13.

Wird nun ein konkreter Wert gesucht[72], so muss bei der Wurzel beginnend stets nur einer der beiden Teilbäume durchsucht werden. Wird der Wert 44 gesucht, so muss nicht die ganze Liste der Werte, beginnend bei 6, durchforstet werden. Vielmehr beginnt es mit dem Lesen des Wertes in der Wurzel. Hierbei handelt es sich um die 16, also einen Wert kleiner als 44. Das heisst: ist 44 im Baum enthalten, dann im rechten Teilbaum der Wurzel.

Geht man in diesen rechten Teilbaum, so stößt man zuerst auf die 35. Wegen $35 < 44$ muss wiederum der rechte Teilbaum des Knotens mit der 35 weiter verfolgt werden. In unserem Beispiel ist der folgende Knoten bereits die 44, wir haben unsere Suche schon nach drei Vergleichen beendet!

[72] Das heisst, es stellt sich die Frage: "Ist ein bestimmter Wert in dem Baum enthalten oder nicht?"

4.3. Elementare Algorithmen

In diesem Abschnitt soll es exemplarisch um einige elementare, zum Teil aber sehr häufig verwendete Algorithmen gehen. Der Begriff *Algorithmus* wurde bereits in Abschnitt 2.1. (vgl. S. 54) definiert. Im wesentlichen handelt es sich um ein Kochrezept, wie ein Problem zu lösen ist.

Zur Darstellung von Algorithmen verwenden wir im Folgenden zum Teil die (auf S. 57 ff vorgestellten) Nassi-Shneidermann-Struktogramme, zum Teil geben wir aber auch einen Quelltext(ausschnitt) in Pascal oder C an.

4.3.1. Vertauschen zweier Speicherinhalte

Beginnen wir mit dem wohl elementarsten Mechanismus: dem Vertauschen zweier Speicherinhalte.

Nehmen wir an, die Variablen `a` und `b` seien von demselben Datentyp (stellvertretend nehmen wir hier ganze Zahlen als Beispiel), dann ist es sehr häufig erforderlich, die Inhalte von `a` und `b` auszutauschen, etwa weil man möchte, dass `a` den kleineren und `b` den größeren Wert beinhaltet.

Dann ist der einfachste Weg, diesen Tausch zu organisieren, eine Hilfsvariable (namens `hilf`) desselben Datentyps bereitzustellen und "im Ring herum" die Werte zuzuweisen:

```
hilf = a;
a = b;
b = hilf;
```

Man kann sich sehr leicht davon überzeugen, dass anschließend `a` und `b` ihre Werte ausgetauscht haben.

4.3.2. Noch einmal: Vertauschen zweier Speicherinhalte

Es gibt aber auch noch eine trickreichere Variante, zwei Speicherinhalte auszutauschen, solange es sich nur um Werte handelt, mit denen gerechnet werden kann. Dann ist sogar ein sogenanntes „*Tauschen am Platz*" möglich, das nachfolgend dargestellt wird. Hierfür ist dann kein dritter Speicherplatz erforderlich. Dafür muss etwas gerechnet (und für die Leserinnen und Leser: nachgerechnet) werden.

```
a = a+b;
b = a-b;
a = a-b;
```

Um nachvollziehen zu können, dass dieser kleine Algorithmus tatsächlich das Gewünschte tut, nehmen wir uns am besten konkrete Werte her und verfolgen Schritt für Schritt die Werte, die in `a` bzw. `b` gespeichert werden.

Speicherbelegung:	a	b
`a = 1;`	1	
`b = 2;`	1	2
`a = a+b;`	3	2
`b = a-b;`	3	1
`a = a-b;`	2	1

Wie man leicht sehen kann, haben `a` und `b` nach der Aktion ihre Werte getauscht.

4.3.3. Primzahlbestimmung durch Division

Häufig muss auch festgestellt werden, ob eine gegebene natürliche Zahl `n` eine Primzahl ist, sich also nur durch 1 und sich selbst ohne Rest teilen lässt.

Nach kurzem Nachdenken stellt man fest, dass man nur testen muss, ob n sich durch eine der Zahlen 2, 3, 4, ... sqrt(n) ohne Rest teilen lässt[73].

Im Struktogramm kann dies so aussehen:

Eingabe: n

Setze i = 2

Solange i < Wurzel aus n und nicht nIstKeinePrimzahl

nIstKeinePrimzahl = i teilt n ohne Rest

i = i+1

Wenn nIstKeinePrimzahl

Ausgabe: "n ist keine Primzahl"

Ausgabe: "n ist Primzahl"

Oder als C-Quellcode:

```
/* i und n seien als int deklariert, n sei eingegeben worden */
i = 2;
nIstKeinePrimzahl = 0;  /* Noch ist nichts bekannt... */

while ( i < sqrt(n)  &&  !nIstKeinePrimzahl )
{
   nIstKeinePrimzahl = (n%i == 0);
   i = i+1;
}
if (nIstKeinePrimzahl)
{
   printf("n ist keine Primzahl");
}
else
{
   printf("n ist eine Primzahl");
}
```

[73] Die Funktion `sqrt(x)` berechnet in den gängigen Programmiersprachen zu einer nichtnegativen Zahl `x` deren Wurzel.

Zur Erläuterung: mit der logischen Variablen `nIstKeinePrimzahl` wird festgehalten, ob bereits bekannt ist, dass `n` keine Primzahl ist. Solange dies nicht der Fall ist, werden über die Variable `i` die Werte von 2 bis Wurzel aus `n` getestet, ob sie Teiler von `n` sind. Mathematisch ist `i` ein Teiler von `n`, wenn `n` ein ganzzahlig Vielfaches von `i` ist, - bzw. wenn `n` geteilt durch `i` den Rest 0 lässt.

So ist 7 ein Teiler von 21, aber 12 ist kein Teiler von 21. Und 23 ist eine Primzahl, denn für die Zahlen 2, 3 und 4 lässt sich sofort nachprüfen, dass weder 2, noch 3, noch die 4 die Zahl 23 teilen. Und damit ist der Test auch schon beendet, denn die Zahl 5 ist nicht mehr kleiner als die Wurzel aus 23 (=4,7958...).

4.3.4. Primzahlbestimmung mit dem Sieb des Eratosthenes

Sind mehrere Zahlen daraufhin zu überprüfen, ob sie Primzahlen sind, oder benötigt man mehrere Primzahlen, dann bietet sich ein weiteres Verfahren an: das sogenannte *Sieb des Eratosthenes*.

In Worten beschrieben funktioniert das Verfahren wie folgt. Angenommen, es werden die Primzahlen von 1 bis n benötigt. Dann startet man mit der Menge aller natürlichen Zahlen von 2 bis n einschließlich. Anschließend entfernt man aus der Menge alle Vielfachen der Zahl 2, belässt die 2 selbst aber in der Menge. Die nächste Zahl - nach der 2 - die jetzt noch in der Menge ist, ist die 3. Im nächsten Durchgang werden also alle Vielfachen der 3 aus der Menge entfernt, die 3 selbst jedoch bleibt wieder darin. Die 4 ist bereits gestrichen worden, also ist die 5 die nächste Zahl, die in diesem Sieb enthalten ist. Alle Vielfachen der 5 - ohne die 5 selbst - werden nun entfernt. Und so geht das Verfahren weiter, bis wir bei der Zahl n angelangt sind.

Etwas übersichtlicher lässt sich dies als Struktogramm formalisieren.

```
sieb := [ 2,3,4,...,n-1,n ]
j := 2
solange j in sieb
    i := 2
    solange i*j <= n
        entferne i*j aus dem sieb
        i := i+1
    j := nächstgrößere Zahl, die noch in sieb ist, falls vorhanden
```

Nach Durchlaufen des Verfahrens[74] enthält die Menge sieb genau die Primzahlen zwischen 2 und n. So ist für n = 20 die Ergebnismenge[75] { 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~ } = { 2, 3, 5, 7, 11, 13, 17, 19 }.

---

[74] Der Algorithmus endet, sobald kein j kleiner oder gleich n mehr in dem Sieb vorhanden ist.

[75] Sofern Sie diese Seite in Farbe sehen: die blau markierten Streichungen sind bei dem Durchgang mit j = 2 erfolgt, die roten Streichungen wurden im nächsten Durchlauf für j = 3 durchgeführt. Danach gab es bereits nichts mehr zu streichen.

4.3.5. Der Euklidische Algorithmus

Ein Beispiel für einen recht einfachen Algorithmus, der jedoch in sehr anwendungsorientierten Zusammenhängen nützlich ist (vgl. Abschnitt 5.4.2.), ist der *Euklidische Algorithmus* zur Bestimmung des *größten gemeinsamen Teilers* (*ggT*) von zwei ganzen Zahlen.

Ohne Einschränkung seien a und b zwei positive ganze, also zwei natürliche Zahlen. Dann lässt sich der Euklidische Algorithmus so beschreiben:

```
WHILE a <> b DO
BEGIN
  IF a > b THEN
    a = a - b
  ELSE
    b = b - a
END
ggt = a
```

Das heißt: allein durch fortgesetzte Vergleiche und Subtraktionen kann der ggT zweier ganzer Zahlen ermittelt werden. (Für negative ganze Zahlen ist lediglich zuvor das Vorzeichen zu entfernen; auf den ggT hat dies keinen Einfluss.)

4.4. Sortierverfahren

In diesem Abschnitt wollen wir einige der zahlreichen Sortierverfahren behandeln, die für die verschiedensten Szenarien entwickelt worden sind. Als Standardreferenz sei wieder auf das klassisch zu nennende Buch von [Wirth] hingewiesen.

Wir gehen in unserem Rahmen davon aus, dass wir ein Array von n Elementen zu sortieren haben. Ohne Einschränkung und zur Vereinfachung der Darstellung gehen wir stets davon aus, dass es sich um ein Array mit ganzen Zahlen handelt, die aufsteigend sortiert werden sollen. Natürlich können in der Praxis auch andere Datensatzstrukturen sortiert werden, in diesem Falle tritt an den hier verwendeten Vergleich zweier ganzzahliger Werte der Vergleich entsprechender Attribute bezogen auf den Ordnungen, beispielsweise Zeichenketten mit der lexikographischen Ordnung[76].

4.4.1. Sortieren durch direktes Einfügen

Unser Array `a` (mit den Werten `a[1], a[2], ..., a[n]`) wird gedanklich in zwei Teile zerlegt. Der Anfang `a[1],...,a[i]` ist die Ziel-, der zweite Teil `a[i+1],...,a[n]` die Quellsequenz. Wir beginnen mit `i=2`, betrachten `a[i]=a[2]` und vergleichen dies mit `a[1]`, ist `a[1]<=a[2]`, so belassen wir beide an ihrem Ort; sollte `a[1] > a[2]` sein, so setzen wir `a[2]` an die Stelle von `a[1]`, das alte `a[1]` rutscht nach `a[2]`. Anschließend wird für `i=3` `a[3]` mit `a[1]` und `a[2]` verglichen und ggf. an

---

[76] Zwei Zeichenketten können *lexikographisch* verglichen werden, indem zunächst das erste Zeichen beider Zeichenketten (s1 und s2) verglichen wird. Kommt das erste Zeichen von s1 gemäß der Codierung (z.B. ASCII) vor dem ersten Zeichen von s2, dann ist s1 < s2. Sind beide erste Zeichen gleich, dann wird entsprechend mit dem zweiten Zeichen [sofern vorhanden] fortgefahren. Insgesamt gelangt man auf diese Weise im wesentlichen auf die Sortierung, die man von Telefonbüchern oder Adressverzeichnissen kennt.

passender Stelle eingefügt. Dergestalt geht es weiter, bis `i=n` und das Verfahren zum Ende gekommen ist.

An einem konkreten Beispiel mit `n=6` illustriert:

Start:	34	23	13	12	27	15
i=2	23	34	13	12	27	15
i=3	13	23	34	12	27	15
i=4	12	13	23	34	27	15
i=5	12	13	23	27	34	15
i=6	12	13	15	23	27	34

Allgemein (in Pascal) formuliert:

```
FOR i:=2 TO n DO
BEGIN
   x := a[i];
   a[0] := x;
   j := i-1;
   WHILE x < a[j] DO
   BEGIN
      a[j+1] := a[j];
      j := j-1
   END;
   a[j+1] := x
END;
```

Anmerkung: die Komponente `a[0]` dient hier nur dem Zweck, dass die `WHILE`-Schleife kompakter formuliert werden kann. Durch die Bedingung `x < a[j]` bricht diese Schleife nämlich spätestens automatisch dann ab, wenn `j=0` erreicht ist.

4.4.2. Sortieren durch direktes Auswählen

Das Gegenstück zum Einfügen ist das Sortieren durch direktes Auswählen. Hier wird jeweils das kleinste Element (des restlichen Arrays) bestimmt und mit dem ersten Element (des Restes) vertauscht.

An einem Beispiel illustriert:

Start:	34	23	13	12	27	15
i=1	**12**	23	13	34	27	15
i=2	**12**	**13**	23	34	27	15
i=3	**12**	**13**	**15**	34	27	23
i=4	**12**	**13**	**15**	**23**	27	34
i=5	**12**	**13**	**15**	**23**	**27**	34

Die fettgedruckten (und bei Farbdarstellung grün dargestellten) Zahlen sind die bereits sortierten Werte, wobei für i=5 selbstverständlich auch der sechste Wert, die 34, automatisch bereits sortiert vorliegt.

Die formale Beschreibung dieses Verfahrens soll den eifrigen Lesern überlassen bleiben. `:-)`

4.4.3. Sortieren durch paarweisen Austausch (Bubblesort)

Unter dem Namen *Bubblesort* ist ein Sortierverfahren bekannt, bei dem sukzessive immer benachbarte Werte nötigenfalls ausgetauscht werden, - solange, bis die Sortierung vollständig ist.

Der Algorithmus, hier wieder in Pascal dargestellt, ist sehr einfach.

```
FOR i:=2 to n DO
BEGIN
   FOR j:=n DOWNTO i DO
   BEGIN
      IF a[j-1] > a[j] THEN
      BEGIN
         hilf := a[j-1];
         a[j-1] := a[j];
         a[j] := hilf
      END
   END
END
```

Dieser Algorithmus kann natürlich noch optimiert werden: beispielsweise kann man sich merken, ob in einem Durchgang überhaupt noch ein Austausch erforderlich war. War dies nicht der Fall, dann ist das Array offensichtlich bereits sortiert.

4.4.4. Quicksort

Als Vertreter eines etwas fortgeschritteneren Verfahrens, das dann auch in seiner Implementierung etwas aufwendiger ist als die bislang vorgestellten, recht einfachen Algorithmen, soll hier der *Quicksort* behandelt werden.

Die Grundidee: das zu sortierende Ausgangsarray wird willkürlich in zwei Teile geteilt, d.h. es wird ein Index i bestimmt; nennen wir das zugehörige Array-Element x = a[i]. Nun wird das Array von links kommend durchsucht bis ein Element a[j] gefunden wird mit a[j] > x. Entsprechend wird das Array von rechts durchsucht bis ein Element a[k] gefunden wird mit a[k] < x. (Sollte es beides nicht geben, dann sind alle Werte im Array gleich, das Array wäre also als Spezialfall schon sortiert; sollte es eines der beiden nicht geben, dann wäre x das Minimum oder das Maximum, das entsprechend einsortiert werden könnte; das Verfahren würde dann mit dem restlichen n-1 Elemente großen Array fortgeführt werden.)

Seien j und k solche Indizes, dann werden a[j] und a[k] miteinander vertauscht. Dies wird fortgeführt, und so erhalten wir schließlich ein Array, in dem links von x nur Werte kleiner als (oder gleich) x stehen, rechts von x sind alle Werte größer oder gleich.

Dieselbe Prozedur kann nun rekursiv auf diese beiden Teilarrays angewendet werden.

Spielen wir dieses Verfahren exemplarisch durch.

Wir starten mit dem folgenden Array, n=7.

```
23 45 66 17 28 90 33
```

Wir wählen willkürlich x=28, also i=5.

```
23 45 66 17 28 90 33
```

Nun wird von links kommend, also bei Index 1 startend, geprüft, ob ein Element a[j] > 28 ist. Dies ist für j=2 der Fall: a[2]=45 > x=28.

Von rechts kommend ist die 17 das erste Element, das kleiner als x=28 ist; der Index hierfür ist k=4.

Anschließend werden diese beiden Elemente vertauscht, j wird auf 3 erhöht, k auf 3 heruntergesetzt.

```
23 17 66 45 28 90 33
```

a[3]=66 > 28, k wird auf 2 heruntergesetzt.

Damit haben wir zwei Teilarrays gefunden: das eine umfasst die beiden Elemente a[1] und a[2], 23 und 17, das zweite den Rest ab a[3]. Alle Elemente des ersten Teilarrays sind kleiner als alle Elemente des zweiten.

Nun kann das Verfahren für jedes der beiden Teilarrays wiederholt werden bis schließlich alle Stücke sortiert sind.

Eine Implementierung des Quicksort kann z.B. in [Wirth] nachgeschlagen oder selbst programmiert werden.

5. KRYPTOLOGIE (KRYPTOGRAPHIE)

5.1. Einführung

Unter *Kryptologie* versteht man die Wissenschaft der Entwicklung von Kryptosystemen, das heißt Mechanismen zur Verschlüsselung von Nachrichten oder allgemeiner Daten; zu diesem Teilaspekt, der häufig auch *Kryptographie* genannt wird, kommt die sogenannte *Krypto-analyse*, also die Forschung nach Möglichkeiten, ein Kryptosystem zu „knacken".

Für unseren Rahmen seien zunächst einmal einige Quellen für ein tiefergehendes Kennenlernen kryptologischer Methoden erwähnt: einmal gibt es von Albrecht Beutelspacher ein kleines Büchlein „Kryptologie" (siehe [Beutelspacher]), das einen sehr hübschen Einstieg in die Thematik gewährt; zum zweiten findet sich im World Wide Web ein umfangreicher Einführungstext von Wolfgang Bachhuber auf dessen eigenem Webserver (siehe [Bachhuber] im Literatur- und Quellenverzeichnis). Auch die Webseiten von [Ahrweiler] liefern eine lesenswerte Einführung in die Thematik. Schließlich sei auch auf das „große" Buch von Beutelspacher u. a. [BeuSchWol] hingewiesen, in dem es sehr ausführlich um die verschiedenen Themen der Kryptographie geht.

Grundsätzlich unterscheidet man Verfahrensweisen der Kryptologie nach verschiedenen Kriterien, vor allem dabei nach der Komplexität des Algorithmus und der Länge des sogenannten Schlüssels (keys). Ein sehr zentrales Merkmal ist aber auch, ob der oder die verwendeten Schlüssel privat oder teilweise öffentlich sind; man spricht von *symmetrischen Kryptoverfahren*, wenn mit demselben (dann natürlich nicht öffentlichen) Schlüssel codiert und decodiert wird. Ein Verfahren ist *asymmetrisch*, wenn es je einen Schlüssel für das Ver- und das Entschlüsseln gibt.

Einige begriffliche Klärungen und Vereinbarungen vorweg. Wir wollen der Einfachheit halber als Nachrichten Texte im Sinne von Zeichenketten betrachten. Der Ausgangstext ist der sogenannte *plain text* (Klartext), die verschlüsselte(n) Version(en) wird (bzw. werden) *cipher text* (Geheimtext) genannt. Wo nichts anderes angegeben wird, wollen wir uns an eine von [Beutelspacher] angeregte Notation[77] halten und mit Kleinbuchstaben den Klar- und in Großbuchstaben den Geheimtext schreiben.

Wir wollen uns im Folgenden mit Standardansätzen der Verschlüsselung beschäftigen um einen Einblick in die Materie zu erhalten. Für tiefergehende Fragen werden an entsprechenden Stellen weiterführende Quellen genannt.

5.2. Symmetrische Kryptosysteme (Private Key)

Betrachten wir zunächst Verschlüsselungsverfahren, bei denen Sender und Empfänger einer Nachricht diese mit der Kenntnis desselben Schlüssels bearbeiten. Daher der Name „symmetrisch". Naheliegenderweise darf diesen Schlüssel kein Außenstehender erfahren.

---

[77] Ohne Einschränkung werden exemplarisch also nur die Kleinbuchstaben chiffriert; für die einfachen Beispiele zu Beginn genügt dies aber für das Verständnis vollauf.

5.2.1. Transpositionsalgorithmen

Ein *Transpositionsalgorithmus* ist ein Verfahren, bei dem die Zeichen innerhalb eines gegebenen Textes vertauscht, mathematisch formuliert: permutiert werden.

Ein simples Beispiel eines solchen Verfahrens ist die *Skytale von Sparta*. (vgl. [Beutelspacher]). Dabei handelt es sich um einen Zylinder, auf dem Pergament spiralförmig aufgerollt werden kann. Je nach Radius des Zylinders (und natürlich der Schriftgröße) kommen andere Zeichenketten zum Vorschein. Ein Beispiel: angenommen, wir haben den folgenden Text linear auf ein Papier oder Pergament geschrieben.

```
whimianarffttttoissrkc
```

Wickeln wir dieses Papier spiralförmig auf einen Zylinder, so dass auf den Umfang jeweils sechs Zeichen passen, dann entsteht folgendes Muster.

```
wntr
hatk
iroc
mfi
ifs
ats
```

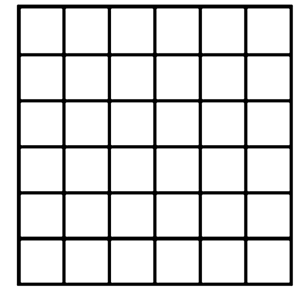
Offensichtlich können wir hier keinen sinnvollen Text erkennen. Hat unser Zylinder jedoch einen Umfang von vier Zeichen, dann entsteht folgende Zeichenmatrix, der wir nun schon eine Information entnehmen können.

```
wirtsc
hafts
infor
matik
```

Offensichtlich ist dieser Algorithmus sehr einfach, denn für das „Knacken" des Geheimtextes müssen wir lediglich die verschiedenen Varianten des „Abrollens" auf einem (gedanklichen) Zylinder durchspielen (oder von einem Computer durchspielen lassen).

Dennoch sind Transpositionsalgorithmen heutzutage sehr wichtige Bausteine für die Chiffrierung von Nachrichten. An dieser Stelle soll noch auf ein weiteres, etwas weniger triviales Verfahren dieser Familie eingegangen werden.

Hierzu nehmen wir uns der Einfachheit halber wieder das Wort „Wirtschaftsinformatik" her. Für den Algorithmus brauchen wir eine Schablone, z.B. die hier gezeigte 6 x 6-Matrix.



Dort können wir nun das gewünschte Wort eintragen.

w		i		r	
		t		s	c
h		a	f		
t		s	i		n
	f	o		r	m
a	t		i		k

Jetzt müssen bloß in die restlichen Felder beliebige Buchstaben eingetragen werden, und schon haben wir einen codierten Text vor uns.

w	a	i	l	r	d
a	m	t	e	s	c
h	i	a	f	s	e
t	n	s	i	i	n
m	f	o	w	r	m
a	t	a	i	l	k

Das Entziffern dieses Geheimtextes dürfte ohne die korrekte Schablone nicht ganz trivial sein, insbesondere dann nicht, wenn in die Leerfelder korrekte Worte oder Satzteile eingetragen werden und auf diese Weise auch so etwas wie die statistische Häufigkeit einzelner Buchstaben realen Texten sehr nahe kommt.

5.2.2. Substitutionsalgorithmen (u.a. DES)

Unter einem *Substitutionsalgorithmus* versteht man ein Verfahren, bei dem jedes Zeichen durch ein anderes ersetzt wird.

Ein sehr elementares Substitutionsverfahren ist die *Verschiebechiffre*. Hier wird jedes Zeichen um eine feste Anzahl im jeweiligen Alphabet (Zeichenvorrat) verschoben.

5.2.2.1. Ein primitives Verfahren: ROT13

Als *ROT13* hat sich ein (primitives) Verfahren etabliert, mit dem man z.B. in Newsgruppen das spontane Lesen oder Erkennen einer Nachricht verhindern möchte. Hierbei wird jeder Buchstabe um 13 Zeichen weitergesetzt, modulo 26 natürlich. Aus A wird somit ein N, aus dem X ein K, aus dem P ein C. (Wir könnten jetzt weitere Beispiele[78] diskutieren...)

`QRAXRAUNRYGWHAT`

Selbstverständlich ist dieser Algorithmus sehr leicht zu knacken. Es gibt - im Rahmen der 26 Buchstaben unseres Alphabetes - 26 solcher Verschiebechiffren (inklusive der vollkommen

[78] Zum Beispiel wäre zu überlegen, ob folgendes zutrifft: IVRYYRVPUGZNPUGRFVUARANHPUFCNFF.

untauglichen Identität), und bezogen auf den ASCII-Code wären es auch nur 256 verschiedene Verfahren, die leicht per trial and error ermittelt werden könnten.

Eines darf hier allerdings nicht außer acht gelassen gewerden: das Erraten eines solchen Verschiebecodes setzt voraus, dass man bewerten kann, wann die „dechiffrierten" Texte einen Sinn ergeben. Dies von einem Computer durchführen zu lassen würde bedeuten, dass dieser Zugriff auf Wörterbücher der in Frage kommende(n) Sprache(n) besitzt.

5.2.2.2. Statistische Betrachtungen

Eine weitere Anmerkung: gerade im Zusammenhang mit Substitutionsalgorithmen gewinnt die Statistik an Bedeutung. Gehen wir von der deutschen Sprache aus, so stellen wir fest, dass gewisse Buchstaben wie etwa „e", „n" oder „s" sehr viel häufiger vorkommen als andere. (Nicht nur das „q" ist recht selten.) So hat das „e" in etwa einen Anteil von 17,4 %, das „n" von knapp 10% und das „s" von über 7% (zitiert nach [Beutelspacher], vgl. auch nachstehende Tabelle).

Buchstabe	Häufigkeit %		Buchstabe	Häufigkeit %
a	6,51		n	9,78
b	1,89		o	2,51
c	3,06		p	0,79
d	5,08		q	0,02
e	17,40		r	7,00
f	1,66		s	7,27
g	3,01		t	6,15
h	4,76		u	4,35
i	7,55		v	0,67
j	0,27		w	1,89
k	1,21		x	0,03
l	3,44		y	0,04
m	2,53		z	1,13

Tabelle: Häufigkeiten der Buchstaben in deutschen Texten (nach [Beutelspacher])

Bei einfachen Substitutionsalgorithmen kann somit über die Verteilungsstatistik eine Annahme über gewisse Zuordnungen gemacht und eventuell sogar der Code auf Anhieb erraten werden.

Eine Chiffrierung muss natürlich nicht einem Buchstaben stets denselben anderen Buchstaben zuordnen. Tut sie es aber, so nennen wir sie *monoalphabetisch*, andernfalls *polyalphabetisch*.

Offensichtlich gibt es gerade $26! = 26 \cdot 25 \cdot 24 \cdot 23 \cdot \ldots \cdot 2 \cdot 1 \sim 4 \cdot 10^{26}$ Möglichkeiten monoalphabetischer Chiffrierungen über der Menge der Kleinbuchstaben { a, b, ... , z }.

Selbstverständlich gibt es eine riesige Anzahl an Varianten von Substitutionsalgorithmen. Durch den Einsatz von Computern müssen viele jedoch sehr schnell als ziemlich unsicher betrachtet werden.

5.2.2.3. Data Encryption Standard (DES)

Die populärste monoalphabetische Chiffrierung ist der von IBM entwickelte *Data Encryption Standard* (*DES*), der seit 1977 standardisiert ist. Mit ihm werden nicht (die 26) Buchstaben sondern 64-Bit-Sequenzen codiert, also Folgen von 64 Nullen und Einsen. Gemäß Standard werden 8 der 64 Bits für Fehlererkennung eingesetzt, so dass für den eigentlichen Schlüssel immerhin noch 56 Bits übrigbleiben, das heißt, dass es 2^{56} verschiedene DES-Schlüssel gibt.

Mit DES sind beispielsweise die Geheimzahlen auf unseren Eurocheque-Karten verschlüsselt.

Eine Variante auf DES ist das sogenannte *Triple DES*, das das klassische DES-Verfahren einfach dreimal hintereinander anwendet: zunächst wird der Klartext mit einem Schlüssel A gemäß DES codiert, dann mit dem Schlüssel B decodiert, dann erneut mit A codiert.

Interessant ist hier ein von dem Holländer Kerckhoffs bereits 1883 formuliertes Prinzip, das den Kern der Problematik gut umreißt[79]: *„Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Algorithmus abhängen. Die Sicherheit gründet sich nur auf die Geheimhaltung des Schlüssels."*

5.2.3. Polyalphabetische Chiffrierungen

Monoalphabetische Chiffrierungen haben den Nachteil, dass sie die statistische Analyse eines Geheimtextes begünstigen, denn korrespondierend zu den Häufigkeitsverteilungen der einzelnen Buchstaben in der jeweiligen Sprache treten in den Geheimtexten diese Häufigkeiten wieder auf: nur vertauscht bei dem jeweiligen Ersatzzeichen des Codes.

Bei polyalphabetischen Chiffrierungen wird ein Zeichen nicht immer mit demselben anderen Zeichen codiert. Daher ist die Analyse der Häufigkeitsverteilungen (von <u>einzelnen</u> Zeichen) nicht mehr erfolgreich.

Beispielsweise kann über folgende mengenmäßige Zuordnung von zweistelligen Zahlen zu Buchstaben dafür gesorgt werden, dass zum einen leicht codiert und decodiert werden kann, dass zum anderen aber die Häufigkeiten der zweistelligen Zahlen (annähernd) gleichverteilt sind. Dazu werden zu jedem Buchstaben seiner statistischen Häufigkeit entsprechend viele Zahlenpaare angeboten. Nachstehend ein Auszug einer solchen Zuordnung.

```
A	09 18 22 52 58 63 89
B	14 82
C	01 08 41
D	07 17 71 72 83
E	03 12 43 46 51 54 61 65 68 73 77 79 80 83 87 90 93
```

[79] Zitiert nach [Beutelspacher], S. 23.

```
F     00 13
G     04 27 88
H     11 45 53 67 78
...
Y     44
Z     16
```

Damit kann dann ein Text sehr einfach codiert werden: für jeden Buchstaben des Klarschrifttextes wird zufällig eine der zugeordneten Zahlen ausgewählt. Aus BACH wird auf diese Weise 14580145. Oder auch 82090167. Und decodiert werden kann er ebenso einfach. (Tun Sie's doch einmal für 44792245...)

Dadurch, dass die Häufigkeiten der einzelnen (Geheim-)Zeichen einigermaßen gleichverteilt sind, ist ein solcher Geheimtext aber natürlich noch lange nicht sicher. So können zum Beispiel aus der Häufigkeitsverteilung von Zwei-Zeichen-Gruppen ebenfalls Schlussfolgerungen gezogen werden.

Der französische Diplomat Vigenère führte 1586 die nach ihm benannte Verschlüsselung ein, bei der verschiedene monoalphabetische Chiffrierungen im Wechsel eingesetzt werden. Hierfür benötigen wir das sogenannte Vigenère-Quadrat und ein Schlüsselwort.

```
abcdefghijklmnopqrstuvwxyz        Klartext
ABCDEFGHIJKLMNOPQRSTUVWXYZ        Das Vigenère-Quadrat
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHI
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
RSTUVWXYZABCDEFGHIJKLMNOPQ
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTUV
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
```

Dieses Quadrat besteht also aus 26 untereinander geschriebenen Alphabeten, die jeweils um ein Zeichen weiter versetzt sind.

Wählen wir als Schlüsselwort KAUKASUS, so können wir einen Klartext wie folgt verschlüsseln. Wir schreiben Schlüsselwort und Klartext übereinander.

Schlüsselwort:	K	A	U	K	A	S	U	S	K	A	U	K
Klartext:	s	t	r	e	n	g	g	e	h	e	i	m

Der Buchstabe des Schlüsselwortes über einem Zeichen des Klartextes legt die Zeile aus dem Vigenère-Quadrat fest, mit dem zu chiffrieren ist.

Über dem „s" steht das „K", also wird mit dem Alphabet in der „Zeile K" chiffriert: zu dem Buchstaben „s" gehört dort „C". Über dem „t" steht das „A", also finden wir im Quadrat in der „Zeile A" zu „t" den Codebuchstaben „T". Und so fahren wir fort und erhalten den Geheimtext „CTLONYAWRECW".

Es muss jedoch angemerkt werden, dass auch dergestalt verschlüsselte Texte nicht unbedingt sicher sind. 1863 wurde von Friedrich Wilhelm Kasiski ein einfacher Test auf die Schlüssellänge publiziert, der jedoch auf eine Idee von Charles Babbage zurückgeht.

Die Idee: treten im Klartext identische Buchstabenfolgen auf, z.B. häufige Worte wie "ein" oder "die", dann entsprechen diesen i.a. unterschiedliche Cipher Text Zeichenfolgen; sind diese identischen Klartextzeichenfolgen jedoch um ein Vielfaches der Schlüssellänge voneinander entfernt, so sind die zugehörigen verschlüsselten Zeichenfolgen identisch!

Der *Kasiski-Test*: liegt ein im Vergleich zum verwendeten Schlüssel langer Cipher Text vor, so kann in letzterem nach Wiederholungen von Buchstabengruppen (-mindestens Paaren-) gesucht werden; die Differenz dieser Wiederholungen ist ein mögliches Vielfaches der Schlüssellänge.

Betrachten wir das nachstehende Beispiel.

```
Schlüssel: FHDW
Text:      nachmittagsfindetdieveranstaltungimaltbaustatt
Codiert:   SHFDRPWPFNVBNUGAYKLAALUWSZWWQAXJLPPWQAEWZZWWYA
           ....|....|....|....|....|....|....|....|....|.
Wiederholung gefunden bei i= 5 und j=34 [PW]   Indexdifferenz: 29
Wiederholung gefunden bei i=25 und j=41 [ZW]   Indexdifferenz: 16
Wiederholung gefunden bei i=26 und j=42 [WW]   Indexdifferenz: 16
Wiederholung gefunden bei i=27 und j=35 [WQ]   Indexdifferenz: 8
Wiederholung gefunden bei i=28 und j=36 [QA]   Indexdifferenz: 8
```

Die Schlüssellänge ist 4, die festgestellten Wiederholungen (von Paaren) ergeben die Werte 29, 16 und 8. Die 29 ist ein typischer "Fehlfund", die 8 und die 16 dagegen deuten jedoch auf eine Schlüssellänge, die in 8 und 16 als Teiler vorkommt, also 8, 4 oder 2.

Bei genauerem Hinsehen stellt man fest, dass die Sequenzen ZWW und WQA sogar Dreierketten sind, und diese wiederholen sich bei einer Differenz von 8 bzw. 16.

Nun kann also mit den Schlüssellängen 2, 4 und 8 probiert werden, den Cipher Text zu knacken. Dabei ist allerdings noch einmal zu betonen, dass diese "Erkenntnis" nur eine Vermutung darstellt, oftmals wird man auch durch zufällige Wiederholungssequenzen auf fehlerhafte Kandidaten für die Schlüssellängen stoßen.

Zur Ergänzung des Kasiski-Testes bietet sich ein auf den ersten Blick komplizierter wirkendes Verfahren, der Friedman- oder Kappa-Test an. Dieser Algorithmus wurde 1952 von William Friedman entwickelt. Ausgangsfrage hierbei: "*Mit welcher Wahrscheinlichkeit besteht ein willkürlich herausgegriffenes Buchstabenpaar aus zwei gleichen Buchstaben?*".

Betrachten wir eine Zeichenkette der Länge n. Seien n_a, n_b usw. die Anzahl der Buchstaben a, b usw. in dem Text. Nun interessieren wir uns für die Anzahl der Paare mit zwei gleichen Buchstaben, wobei diese nicht direkt hintereinander stehen müssen. Das erste a beispielsweise kann eines von n_a vielen sein; für das zweite a bleiben dann n_a-1 Stück zur Auswahl. Die Reihenfolge spielt hierbei keine Rolle, so dass es $\frac{n_a(n_a-1)}{2}$ Möglichkeiten gibt.

Somit ist die Anzahl der Paare, bei denen beide Buchstaben gleich sind: $\frac{n_a(n_a-1)}{2} + \frac{n_b(n_b-1)}{2} + \ldots = \sum_{a \text{ aus } a..z} \frac{n_a(n_a-1)}{2}$

Die Wahrscheinlichkeit, ein Paar aus zwei gleichen Buchstaben zu "ziehen", ergibt sich somit zu[80]

$$\kappa = \frac{\sum_{a \text{ aus } a..z} \frac{n_a(n_a-1)}{2}}{\frac{n(n-1)}{2}}$$

Diese Zahl heißt der *Friedmansche Koinzidenzindex* κ (lies: "kappa").

Kennen wir die Verteilung der einzelnen Buchstaben (z.B. innerhalb der deutschen Sprache), und seien dies die Wahrscheinlichkeiten bzw. relativen Häufigkeiten p_a, ..., p_z für die Buchstaben a, ... , z, dann ergibt sich als Wahrscheinlichkeit dafür, ein Paar aus zwei gleichen Buchstaben zu ziehen, der (gerundete[81]) Wert $\sum_{a \text{ aus } a..z} p_a^2$.

Für einen Text der deutschen Sprache ergibt sich (vgl. hierzu die Tabelle auf S. 99) dieser Wert zu 0,0762. Wären die Buchstaben dagegen völlig gleichverteilt, d.h. wäre $p_a = p_b = \ldots = p_z = 1/26$, so ergäbe sich der Wert $\sum_{a \text{ aus } a..z} p_a^2 = 0,0385$. (Dies ist übrigens der kleinste Wert, den dieser Ausdruck unter der Nebenbedingung $\sum_{a \text{ aus } a..z} p_a = 1$ annehmen kann[82].)

Wurde ein Klartext mit einem monoalphabetischen Verfahren verschlüsselt, so wird auch der Cipher Text denselben Koinzidenzindex wie der Klartext aufweisen, d.h. näherungsweise wiederum 0,0762 im Falle eines "typischen" deutschsprachigen Textes. Ist dagegen der Koinzidendindex eines Cipher Textes deutlich kleiner als dieser Wert, dann wird er vermutlich nicht monoalphabetisch verschlüsselt worden sein.

5.2.4. One Time Pad

1917 wurde von Gilbert Vernam das Verfahren *one time pad* vorgestellt. Hierbei wird - analog zum Vigenère-Mechanismus - mit sehr vielen Schlüsseln gleichzeitig chiffriert. Gehen wir von einer Klartextlänge n aus, dann wird auch ein Schlüssel der Länge n zufällig aus der Menge aller möglichen Schlüssel gewählt. Betrachten wir die 26 üblichen Buchstaben, dann gibt es 26^n verschiedene Schlüsselfolgen. Jedes Zeichen des Klartextes wird nun mit dem betreffenden Schlüsselzeichen codiert.

Betrachten wir das Verfahren auf einzelnen Bits und legen die Modulo-2-Addition[83] zugrunde, dann kann die Chiffrierung schematisch wie folgt dargestellt werden.

---

[80] Dies ist die Anzahl der gewünschten Möglichkeiten dividiert durch die Anzahl aller Möglichkeiten.

[81] Streng formal wird hier zugelassen, dass zweimal dieselbe Position aus dem Text ausgewählt wird. Für große n ist dies aber vernachlässigbar.

[82] Mathematisch ambitionierte Leser dürfen diese Aussage gerne formal beweisen.

[83] Das bedeutet: 0+0=0, 0+1=1+0=1, 1+1=0. Technischer orientiert kann dies auch als „XOR" gelesen werden.

Schlüssel:	0	1	1	0	0	0	1	0	0	1	1	1	0
Klartext:	1	1	0	1	1	1	1	0	0	1	0	1	0
Geheimtext:	1	0	1	1	1	1	0	0	0	0	1	0	0

Betrachtet man die „statistische Sicherheit" dieses Verfahrens, dann stellt man fest: alle (0,1)-Folgen der Länge n sind gleich wahrscheinlich. In diesem Sinne hält ein derart codierter Geheimtext den zuvor diskutierten statistischen Analyseversuchen stand[84].

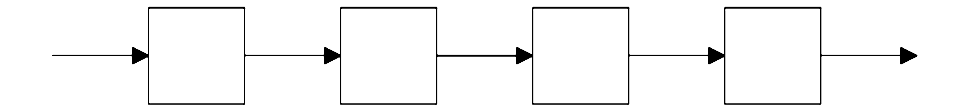
Aber es gibt - natürlich - einen gravierenden Nachteil: da der jeweilige Schlüssel nicht gut zusammen mit der Nachricht verschickt werden kann, muss er entweder zeitlich vorher und/oder auf einem anderen (sicheren!) Kommunikationsweg vom Sender zum Empfänger gelangen oder gelangt sein. Das ist zwar realisierbar, kommt aber für bei weitem nicht alle praktischen Anwendungsfelder in Frage. Insbesondere kann ein solches Verfahren nicht oder nur sehr bedingt im Kontext von Electronic Commerce und Internet eingesetzt werden, da sich die möglichen Kommunikationspartner vor dem Handelsgeschäft nicht (unbedingt) kennen, also auch nicht vorher einen Schlüssel ausgetauscht haben (können).

5.2.5. Schieberegister

Beim interessanten Ansatz des One Time Pads haben wir gesehen, dass ein Problem in der (sicheren) Übermittelung des (langen) Schlüssel(texte)s besteht. Ein wichtiger Ansatz, dieses Problem zu lösen, besteht in der Verwendung pseudozufälliger Zeichen- oder Bitfolgen. Darunter versteht man eine Folge, die nicht wirklich zufällig ist, sich faktisch durch die Kenntnis weniger Daten rekonstruieren lässt, die sich aber in Hinsicht auf zahlreiche Prüfverfahren wie eine zufällige Folge verhält.

5.2.5.1. Lineare Schieberegister

Es gibt zahlreiche Möglichkeiten, Pseudozufallsfolgen zu generieren; in der Praxis haben sich sogenannte *Schieberegister* durchgesetzt, weil diese auch hardwaremäßig gut realisiert werden können und die zugehörige mathematische Theorie sehr weit fortgeschritten ist.



Ein Schieberegister (im obigen Bild ist eines der Länge 4 dargestellt) ist eine endliche Folge von Zellen, die zu einem Zeitpunkt jeweils ein Bit, also 0 oder 1, aufnehmen können. Mit einer bestimmten Taktung werden die Bits "durchgeschoben", am Ausgang des Schieberegisters entsteht auf diese Weise eine Ausgabe-Folge.

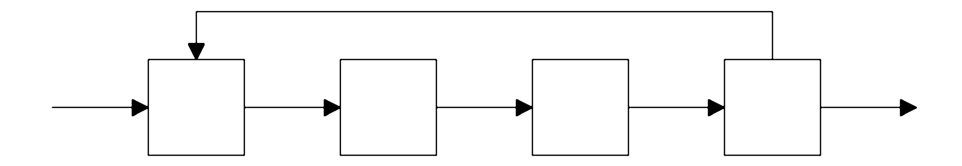
Hierbei ist noch zu klären, nach welcher Gesetzmäßigkeit am Anfang des Schieberegisters welche Bits nachgeliefert werden.

---

[84] [Beutelspacher] legt in seinem Buch dar, dass das One Time Pad ein „perfektes" Chiffrierungssystem ist.

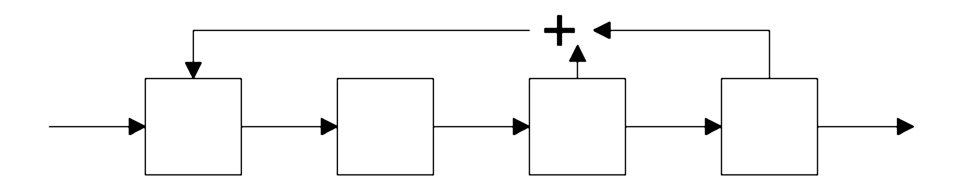
Beispiel: Sei S ein solches Schieberegister mit dem Startzustand (1,0,1,1); dann ist nach einem Takt die Belegung des Registers (-,1,0,1), als Output wurde die 1 generiert. Im nächsten Takt folgt eine weitere 1, das Register besitzt die Belegung (-,-,1,0). [Die Striche "-" deuten an, dass wir noch keinen Mechanismus formuliert haben, wie "von links" die Anfangszellen neue Werte erhalten sollen.

Erster Ansatz: wir können postulieren, dass das Schieberegister den letzten Wert nicht nur nach außen abgibt, sondern diesen auch in die erste Zelle wieder einspeist.



Mit der Anfangsbelegung S(0) = (1,0,1,1) ergeben sich dann die Folgebelegungen S(1) = (1,1,0,1), S(2) = (1,1,1,0), S(3) = (0,1,1,1), S(4) = (1,0,1,1). Offensichtlich ist dieser Mechanismus nicht besonders spannend, denn nach vier Takten stellt sich wiederum die Anfangsbelegung ein, womit auch die Output-Folgen eine Periode von (nur) vier aufweisen. Die früheren Betrachtungen machen klar, dass dies eine gar zu triviale Gesetzmäßigkeit für einen möglichen Schlüssel darstellt!

Der Ansatz der Rückkopplung an sich ist indes in Ordnung, lediglich der Algorithmus darf nicht so elementar sein wie im eben vorgestellten Beispiel.



Die hier gezeigte Skizze soll bedeuten, dass bei jedem Takt die dritte und die vierte Zelle ihre Werte im Sinne der Modulo-2-Addition an die Zelle 1 rückkoppeln.

Damit gilt für die Anfangsbelegung S(0) = (1,0,1,1): S(1) = (0,1,0,1), S(2) = (1,0,1,0), S(3) = (1,1,0,1), S(4) = (1,1,1,0), S(5) = (1,1,1,1), S(6) = (0,1,1,1), S(7) = (0,0,1,1), S(8) = (0,0,0,1), S(9) = (1,0,0,0), S(10) = (0,1,0,0), S(11) = (0,0,1,0), S(12) = (1,0,0,1), S(13) = (1,1,0,0), S(14) = (0,1,1,0), S(15) = (1,0,1,1) = S(0). Die Periode ist somit 15. Der hierbei erzeugte Output ist 110101111000100 [von links nach rechts chronologisch angeordnet].

Dieses Beispiel illustriert, dass lineare Schieberegister konstruiert werden können, mit denen man auf recht einfache Weise Pseudozufallszahlenfolgen[85] generieren kann. Weitergehende Betrachtungen zeigen allerdings, dass die Kryptoanalyse linearer Schieberegister nicht besonders schwierig ist; so kann gezeigt werden, dass ein lineares Schieberegister der Länge

---

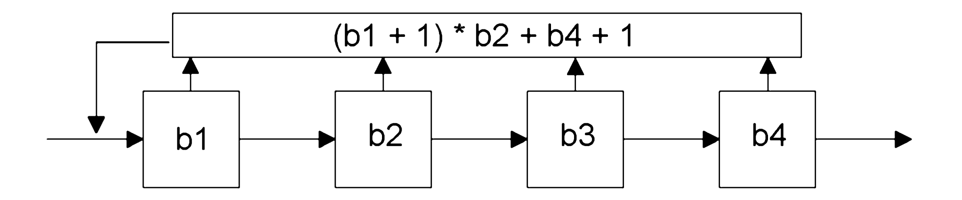
[85] An dieser Stelle gehen wir nicht auf die für Pseudozufallszahlen geforderten mathematischen Rahmenbedingungen ein.

n bereits aus 2n Paaren aufeinanderfolgender Klartext-/Geheimtextbits rekonstruiert werden kann!

5.2.5.2. Nichtlineare Schieberegister

Um dieser Schwachstelle zu begegnen, erweitert man das Konzept der Schieberegister und wird “nichtlinear”. Das heißt: als Rückkopplungsfunktionen werden nicht mehr nur lineare Summenterme verwendet, alle Zellen können auch über andere Funktionsvorschriften auf die Neubelegung der ersten Zelle einwirken.

Hierzu ein konkretes Beispiel.



Im obigen Bild ist ein solches nichtlineares Schieberegister dargestellt. Die Zellen 1, 2 und 4 tragen mit ihrer aktuellen Belegung zur Neuberechnung über die genannte Formel bei zur Neubelegung der Zelle 1 im nächsten Takt.

Hat dieses Schiebregister die Anfangsbelegung S(0) = (0,1,0,1), so ist S(1) = (1,0,1,0), denn b1-neu berechnet sich gemäß der Formel[86] b1-neu = (b1+1)*b2 + b4 + 1 = (0+1)*1 + 1 + 1 = 1.

Dass dieses Schieberegister nicht “versehentlich” wieder linear ist, sieht man daran, dass die Anfangsbelegung (0,0,0,0) nicht auf sich selbst abgebildet wird, d.h. auch wenn mit dem Nullzustand begonnen wird, nimmt das Schieberegister in den folgenden Takten nichttriviale Belegungen an.

5.3. Integrität und Authentikation

Ging es bislang um die Betrachtung passiver Angriffe auf Informationen (Nachrichten), d.h. das Bemühen, Texte zu entschlüsseln, Codes zu knacken, so soll es nun um aktive Angriffe gehen: damit bezeichnen wir das aktive Eingreifen in den Nachrichtenaustausch selbst.

Drei Arten solcher aktiven Angriffe werden üblicherweise unterschieden. Einmal ist zu klären, ob eine empfangene Nachricht unverändert geblieben ist; hier spricht man von *Nachrichtenintegrität*. Die zweite Fragestellung: stammt die Nachricht auch wirklich vom angegebenen Sender? Dies betrifft die *Nachrichtenauthentikation*. Und schließlich bezeichnet *Benutzerauthentikation* das Problem, ob eine Person ihre Identität “beweisen” kann.

---

[86] Natürlich wird hierbei jeweils modulo 2 gerechnet.

5.3.1. Message Authentication Code (MAC)

Zunächst soll es um Nachrichtenintegrität und -authentikation gehen. Hierzu benötigt der Empfänger neben der Nachricht m selbst zusätzliche Information zur Prüfung. Eine solche Information(smenge) wollen wir *Message Authentication Code*, *MAC*, nennen.

Skizzenhaft dargestellt: Wird die Nachricht m ("message") übertragen, so wird mit einem Algorithmus A, der von einem Schlüssel k ("key") abhängt, dieser MAC generiert, kurz mit MAC := A(k,m) notiert[87].

Mit der Nachricht m wird vom Sender auch der ermittelte MAC mitgeschickt; der Empfänger, der ebenfalls den Schlüssel k kennen muss, prüft dann, ob MAC = A(k,m) gilt. Hat ein aktiver Ein- und Angreifer die Nachricht m geändert (und evtl. auch den MAC-Wert), so erhält der Empfänger die (geänderte) Nachricht m' zusammen mit dem (hoffentlich ungültigen!) MAC'. Sofern der Algorithmus A "gut" und der Schlüssel k geheim (geblieben) ist, wird MAC' nicht mit A(k,m') übereinstimmen. Somit erhält der Empfänger das Signal: diese Nachricht ist nicht im Originalzustand - bzw. stammt nicht vom angegebenen Absender.

Selbstverständlich kann der Empfänger die Originalnachricht m nicht wiederherstellen; ggf. muss der Absender die Nachricht m erneut senden.

Zur Berechnung eines solchen MACs werden häufig *Cipher Block Chaining* Mechanismen eingesetzt: Hierbei wird eine Blockgröße n, z.B. 64 Bit, festgelegt. Die Nachricht m (-hier wieder aufgefasst als 0-1-Folge-) wird zerlegt in n-Bit-Pakete m_1, m_2, ... Dann wird mit einer Verschlüsselungsfunktion f der erste Block codiert zu $c_1 := f(m_1)$. c_1 ist also der erste Cipher Text Block. Bevor jetzt m_2 codiert wird, wird dieser Block c_1 bitweise zu m_2 addiert, dann erst wird $c_2 := f(c_1+m_2)$ berechnet. Dies wird wiederholt, bis alle Blöcke sukzessive codiert worden sind.

$$m_1 \overset{f}{\to} c_1;\quad c_1+m_2 \overset{f}{\to} c_2;\quad c_2+m_3 \overset{f}{\to} c_3;\quad \ldots$$

Damit dieser Algorithmus geeignet ist, müssen zwei Forderungen erfüllt sein.

Erstens muss es "praktisch unmöglich" sein[88], zu einem gegebenen MAC-Wert eine passende Nachricht m zu finden. Gilt diese Eigenschaft, so spricht man von einer *Einweg-Hashfunktion*.

Ebenso sollte es "praktisch unmöglich" sein, zwei verschiedene Nachrichten zu finden, die denselben MAC-Wert besitzen. Ist dies erfüllt, so spricht man von einer *kollisionsfreien* Hashfunktion.

Der in Abschnitt 5.2.2.3. kurz vorgestellte DES-Algorithmus (sh. Seite 100) ist ein Beispiel für eine solche Hashfunktion und wird oft für MAC-Berechnungen herangezogen.

---

[87] Ob hierbei die betreffende Nachricht m verschlüsselt ist oder nicht, das spielt hier keine Rolle; an dieser Stelle geht es nicht um Geheimhaltung, sondern um Sicherung der Nachricht.

[88] Mit "praktisch unmöglich" ist gemeint, dass die entsprechende Berechnung "viel zu lange" dauern würde, z.B. mehrere Jahrhunderte. In der Praxis kommt hinzu, dass eine entsprechende Attacke in sehr begrenzter Zeit stattfindet, so dass auch ein Zeitraum von mehreren Wochen oder Monaten meistens ausreichend Schutz bietet.

5.3.2. Benutzerauthentikation

Ebenso bedeutsam wie die Integrität und Authentizität von Nachrichten sicherzustellen ist die zweifelsfreie Identifikation von Personen (Benutzern). Grundsätzlich lässt sich ein Mensch erkennen durch a) bestimmte, charakteristische Eigenschaften, b) durch Besitz (z.B. von Ausweisen) sowie c) durch Wissen[89].

Der erstere Typus ("durch Eigenschaften") spielt zwischenmenschlich eine sehr bedeutende Rolle, wird zur Zeit in Zusammenhang mit technischen Lösungen jedoch auf wenige Merkmale reduziert[90].

Dagegen sind "Wissen" und "Besitz" häufig eingesetzte Verfahren.

Zur Gruppe "Wissen" gehören beispielsweise alle Varianten von Passwörtern. Diese können, sehr einfach, als Klartexte abgespeichert sein, und werden mit einer Benutzereingabe verglichen. Passwörter können aber ebenfalls, wie zuvor beschrieben, über Einweg-Hashfunktionen verschlüsselt worden sein, so dass die Benutzereingabe (im Klartext) entsprechend verschlüsselt und dann mit dem abgelegten Vergleichsmuster abgeglichen wird.

Am Rande sei bemerkt, dass in der Praxis Passwortverfahren vor allem an den menschlichen Schwächen der menschlichen Benutzer kranken. Entweder suchen sich die Benutzer einfachste Passwörter aus, die sie sich gut merken können ("Eva", "Köln", "Mecklenburg-Vorpommern" oder "FHDW"), oder komplizierte Passwörter werden auf Zettelchen geschrieben und auf den Monitor geklebt, womit zumindest gewisse Benutzergruppen einfaches Spiel haben.

Unter dem Strich genügt aber kein Passwortsystem gehobenen Sicherheitsansprüchen, denn selbst wenn die Systemverwaltung die Benutzer zwingt, in festen Zeitabständen ihr Passwort zu ändern, so werden dennoch dieselben Passwörter für mehrere Logins verwendet. Und ein solcher Login kann von einem Angreifer abgehört werden, womit sich dieser anschließend zumindest in der Frist bis zur nächsten Passwortänderung unter der fremden Benutzerkennung anmelden könnte[91].

Damit dieses Problem gelöst werden kann, müssen sich die Daten, die zwischen Benutzer und Rechner ausgetauscht werden, jedesmal ändern. Und natürlich darf nur der berechtigte Benutzer die korrekte Antwort geben können.

Ein solcher Mechanismus wird *Challenge and Response* genannt: der Rechner und der Benutzer verfügen über eine Einweg-Hashfunktion f und einen gemeinsamen, geheimen Schlüssel[92] k. Der Rechner erhält die Benutzerdaten (beispielsweise die Benutzerkennung

---

[89] Auf der Identifikation durch Wissen beruhen die sog. Geheimfragen, die z.B. Mail-Anbieter neben einer gewöhnlichen Passwortabfrage verwenden. "Wie lautet der Geburtsname der Mutter?" - Das ist eine Frage, die nicht nur davon ausgeht, dass die Mutter verheiratet ist und hierbei auch einen anderen Namen angenommen hat, sie geht vielmehr auch davon aus, dass dieser Name keinem Dritten bekannt ist. Was nicht zwingend zutreffen muss...

[90] Beispiele hierfür sind Fingerabdruckskontrollen oder andere biometrische Verfahren wie die Gesichtsanalyse.

[91] Ein noch besserer Sicherheitsstandard wird erreicht, wenn Einmal-Passwörter verwendet werden, also Passwörter, die jeweils nur einmal gültig sind und dann verfallen. Naturgemäß steigt auch der Verwaltungsaufwand hierbei. Dieses Prinzip wird beim Online-Banking mit der Verwendung von Transaktionsnummern (TAN) realisiert.

[92] Selbstverständlich, wie nachfolgend auch weiter beschrieben wird, gibt es für jeden Benutzer einen eigenen Schlüsselwert k.

"user43") und ermittelt damit (etwa aus einer Systemtabelle oder aus einem anderen, globalen Schlüssel errechnet) den Benutzerschlüssel k. Für die folgende Diskussion sei k der korrekte, beim Rechner verwaltete Schlüssel, k' sei der Wert, den der Benutzer (oder der Angreifer) verwendet. Korrekt ist das Ganze, wenn k = k' ist.

Beim Anmeldevorgang sendet der Rechner dem Benutzer (als "Challenge") eine (ganzzahlige Pseudo-)Zufallszahl x. Dieser antwortet, indem er (mit dem Schlüssel k') die Antwort ("Response") f(k',x) berechnet und an den Rechner sendet. Der Rechner ermittelt parallel dazu mit dem Schlüssel k den Wert f(k,x) und vergleicht anschließend, ob f(k,x) = f(k',x) gilt. In diesem Falle scheint alles korrekt zu sein, und der Rechner gewährt dem Benutzer den erwünschten Zugang.

Der große Vorteil eines solchen Verfahrens ist, dass tatsächlich ein konkretes "Passwort" auf diese Weise nur einmal über die Leitung gesendet wird, ein Abhören der Leitung führt also nicht zu einem direkt verwertbaren Zugang[93].

Allerdings muss registriert werden, dass Berechnungen mittels Einweg-Hashfunktionen nicht gut im Kopf stattfinden können, so dass auch auf Benutzerseite ein Computer beteiligt werden sollte, - und ein solcher Computer (genauer: Chip) findet sich auf modernen Chipkarten: die Berechnung der hier benötigten Zahlenwerte findet auf der Chipkarte statt.

Damit "vererbt" sich eine Problematik: woher "weiss" die Chipkarte, dass ihr rechtmäßiger Besitzer sie verwendet? - Auch hier muss dann wieder eine Authentifizierung erfolgen, z.B. mittels einer Geheimnummer (wie bei Bank- oder Kreditkarten).

5.3.3. Zero Knowledge Protokolle

Abschließend wollen wir eine besonders eigenwillige Fragestellung behandeln. Es geht um die Frage, ob eine Person A eine zweite Person B davon überzeugen kann, dass sie ein bestimmtes Geheimnis besitzt, ohne jedoch über dieses Geheimnis etwas zu verraten. Ein solches Verfahren ("Protokoll") wird suggestiv *Zero Knowledge Protokoll* genannt.

Ein historisches Beispiel[94]: Der Mathematiker Niccolò Tartaglia entdeckte 1535 die Methode zur allgemeinen Lösung der kubischen Gleichung $ax^3+bx^2+cx+d = 0$. Die Lösung wollte er jedoch niemandem verraten. - Um Dritte zu überzeugen, dass er die Lösung jedoch kannte, konnte er anbieten, dass ihm Werte für a, b, c und d - also konkrete kubische Gleichungen - vorgelegt wurden. Wenn er immer (und zügig) die korrekten Lösungen angeben konnte[95], dann war es praktisch bewiesen, dass er die Formel besitzen musste[96].

---

[93] Es sei nur am Rande erwähnt, dass natürlich bei entsprechender "Hoheit" über das Netzwerk ein Eindringling die Kommunikation zwischen Sender und Empfänger unterbrechen könnte; in diesem Falle könnte der Angreifer dem Sender (Benutzer) eine Fehlermeldung senden ("Rechner meldet Fehler!") und selbst die Verbindung mit dem Rechner fortführen!

[94] Wie manches andere Zitat ist auch dieses Beispiel dem Buch von [Beutelspacher] entnommen.

[95] Selbstverständlich ist es elementar nachzuprüfen, ob drei konkrete Werte eine bestimmte kubische Gleichung lösen, wohingegen das Finden dieser Lösungen für beliebig viele kubische Gleichungen ohne Kenntnis der Formel von Tartaglia praktisch unmöglich ist.

[96] Die Formel zur Lösung kubischer Gleichungen heißt *Cardanosche Formel*, - benannt nach Geronimo Cardano, der Tartaglia die Formel "abluxte" und sie dann auch veröffentlichte.

1986 stellten die israelischen Wissenschaftler Fiat und Shamir ein (Zero-Knowledge-)Protokoll vor, das überraschenderweise Fiat-Shamir-Protokoll genannt wird. Dieses Protokoll beruht auf der Tatsache, dass es extrem schwierig ist, Quadratwurzeln modulo n einer Zahl v zu finden. Das heißt: Es sei n eine (in der Praxis "große") natürliche Zahl, die keine Primzahl ist. Ist die Primfaktorzerlegung von n unbekannt, dann ist es in begrenzter Zeit nahezu unmöglich, eine Zahl s zu finden, für die $v = s^2 \mod n$ gilt. (s heißt dann eine *Quadratwurzel von v modulo n.*)

Beispiel: Nehmen wir (zur Demonstration) eine "kleine" Zahl n; es sei n = 35. Dann ist hier die Primfaktorzerlegung natürlich bekannt, die Faktoren sind 5 und 7. Suchen wir zu v = 11 die bzw. eine Quadratwurzel modulo n, so kommen wir nach wenigen Versuchen (?) auf die Zahl s = 19, denn 19*19 = 361 = (350+11) = 11 mod 35.

Für die Authentifizierung eines Benutzers (einem Computer gegenüber) wird dieser Mechanismus wie folgt eingesetzt. Da das Hantieren mit sehr großen Zahlen "unmenschlich" ist, wird ein Chip (auf einer Chipkarte) eingesetzt. Der Benutzer erhält (von einer zentralen Instanz, die z.B. die Chipkarte ausgibt) eine Zahl s, die das "Geheimnis" des Benutzers verkörpert. Als öffentlich bekannte Zahl n wird ein Produkt aus zwei "großen" Primzahlen p und q genommen, wobei p und q nicht öffentlich bekannt sind und die Faktorisierung von n, also die Aufspaltung in die Primfaktoren p und q, als "praktisch unlösbar" gilt. Ebenso wird

(5.3.3.1) $v = s^2 \mod n$

berechnet; v ist die öffentliche Identifizierung für den betreffenden Benutzer.

Nochmals im Überblick: p und q sind (geheimgehaltene) "große" Primzahlen, n ist das Produkt dieser beiden und öffentlich bekannt. s ist eine zu n teilerfremde Zahl, die nur dem Benutzer bekannt ist. v wiederum ist öffentlich bekannt.

Der Mechanismus des Fiat-Shamir-Protokolls funktioniert nun wie folgt. Der Benutzer wählt zufällig eine Zahl r, die teilerfremd zu n ist, und berechnet

(5.3.3.2) $x = r^2 \mod n$.

Diese Zahl x wird an den Rechner geschickt, der sich von der Identität des Benutzers überzeugen muss.

Der Rechner wählt zufällig ein Bit b, also 0 oder 1. Ist dieses 0, so soll der Benutzer $y := r \mod n$ berechnen, ist das Bit 1, so soll $y := rs \mod n$ bestimmt werden. Dieser Wert y wird anschließend dem Rechner zurückgeschickt.

Dieser kann nun überprüfen, dass im Falle b=0 die Beziehung $y^2 \mod n = x$ gilt, und dass im Falle b=1 die Gleichung $y^2 \mod n = xv \mod n$ ist.

Begründung: 1.Fall, b=0: $y = r \mod n$; nach (5.3.3.2) ist $y^2 \mod n = r^2 \mod n = x$.
2.Fall, b=1: Hier ist $y := rs \mod n$. Damit ergibt sich: $y^2 \mod n = (rs)^2 \mod n = r^2s^2 \mod n = r^2 \mod n * s^2 \mod n$, und dies ist nach (5.3.3.1) und (5.3.3.2) identisch mit $xv \mod n$.

Anmerkung: Die scheinbare Kompliziertheit, dass der Rechner ein solches Bit b wählen muss, ist dadurch begründet, dass es leicht möglich wäre, jeweils einen solchen Wert r für den ersten oder den zweiten Fall zu finden, d.h. ein Betrug wäre leicht möglich. Dadurch, dass der Authentikationsprozess jedoch dieses zufällige Element besitzt, kann kein Angreifer

vorhersagen, welches Bit ermittelt wird, d.h. auf welchen der beiden Fälle er sich einstellen muss[97].

Ein konkretes Beispiel: Sehen wir uns den Mechanismus des Fiat-Shamir-Protokolls zur Illustration an einem (natürlich wieder einfach gehaltenen) Zahlenbeispiel an.

Es seien p:=19, q:=23 und damit n=437. Der Wert s werde gewählt als s:=305. (305 ist teilerfremd zu n=437.) $v=s^2$ ergibt sich damit zu $v=s^2=381$. (Denn: 305*305 = 93025 = 212*437+381.)

Nun wählt der Benutzer eine Zahl r, die teilerfremd zu n=437 ist. Wir wählen r:=107. Damit ist nach (5.3.3.2) x = 107*107 mod 437 = 11449 mod 437 = 87.

Dieser Wert x wird an den Authentikationsrechner geschickt, der nunmehr ein Bit b "auswürfelt". Nehmen wir an, b wäre 1.

Damit wird der Benutzer aufgefordert, y := rs mod n zu ermitteln und senden. Hier ist das: y = 107*305 mod 437 = 32635 mod 437 = 297.

Jetzt überprüft der Rechner, ob y^2 mod n = xv mod n gilt:

y^2 mod 437 = 297*297 mod 437 = 88209 mod 437 = 372; entsprechend ist x*v mod 437 = 87*381 mod 437 = 33147 mod 437 = 372. - Aha.

Hätte der Rechner dagegen das Bit b=0 ermittelt, dann wäre der Benutzer aufgefordert worden, y := r mod n zu berechnen. Hier: y = 107 mod 437 = 107. Diesen Wert erhält der Rechner, und der berechnet y^2 mod n = r^2 mod n = 87 = x. (In diesem Falle war diese Rechnung trivial, denn y und r waren identisch!)

5.4. Asymmetrische Verfahren (Public Key)

5.4.1. Allgemeines zu asymmetrischen Algorithmen

Die bisher behandelten symmetrischen Verfahren haben zwei wesentliche Eigenschaften:

1. Sender und Empfänger nutzen denselben Schlüssel, können also im Prinzip jeweils sowohl Ver- als auch Entschlüsseln.
2. Die zwei Partner müssen über irgendeinen Kommunikationsweg den gemeinsamen Schlüssel austauschen (können).

1976 wurde von Diffie und Hellman [DifHel76] vorgeschlagen, von diesen Eigenschaften abzuweichen und asymmetrische Algorithmen zu verwenden. Dadurch entfällt die Problematik des notwendigen sicheren Schlüsseltausches, insbesondere im Rahmen der weltweiten Computer- und Kommunikationsvernetzung ein wichtiger Aspekt. Es ist schließlich nicht immer möglich, mit einem Einschreibebrief einem Empfänger einer elektronischen Mail zuvor einen Schlüssel „auf sicherem Wege" zuzustellen!

---

[97] Dieser Mechanismus wird in [Beutelspacher] (S. 95ff) sehr schön an einem sog. "Quadratwurzelspiel" illustriert.

Nachfolgend wollen wir davon ausgehen, dass jeder Teilnehmer T eines asymmetrischen Kryptoverfahrens zwei Schlüssel besitzt: einen öffentlichen (public key) E_T (=encrypt) zur Verschlüsselung und einen geheimen privaten D_T (=decrypt) zur Entschlüsselung.

Wichtig ist, dass der Schlüssel D_T durch die Kenntnis von E_T nicht hergeleitet werden kann.

Die Public Keys solcher Verfahren sind in öffentlich zugänglichen Dateien, im Rahmen des Internets auf sogenannten Schlüssel-Servern, frei zugänglich. Die Private Keys sind natürlich nur dem jeweiligen Besitzer bekannt (bzw. liegen vermutlich auf einem hoffentlich nur dem Besitzer zugänglichen Speichermedium).

Drücken wir das Chiffrieren (E) und Dechiffrieren (D) formelhaft aus, dann muss für ein asymmetrisches Verfahren gelten:

Für jede Nachricht m (message) ist **D(E(m)) = m**.

Für ein asymmetrisches Signaturschema, mit dem beispielsweise elektronische Mails lediglich unterschrieben (signiert) werden sollen, muss dagegen „nur" gelten, dass mittels des öffentlichen Schlüssels E überprüft werden kann, ob bzw. dass m und D(m) „zusammenpassen". Dies kann im Zweifelsfall auch durch Berechnung einfacher Prüfziffern etc. realisiert werden.

Soll eine Nachricht von Teilnehmer A an Teilnehmer B gesandt werden, so besorgt sich A den öffentlichen Schlüssel E_B von B, verschlüsselt die Nachricht m damit und schickt $E_B(m)$ an B. Dieser wiederum ist nach Voraussetzung der einzige, der den privaten Schlüssel D_B besitzt und somit die Nachricht wieder erlangen kann: $D_B(E_B(m)) = m$.

Es ist anzumerken, dass hierbei nur die beiden Schlüssel des Teilnehmers B zum Einsatz kommen, und dass kein geheimer Austausch von Schlüsseln (über irgendeinen als sicher eingestuften Kommunikationskanal) erforderlich ist!

5.4.2. RSA: Mathematische Grundlagen

1977 kreierten Ronald *R*ivest, Adi *S*hamir und Leonard *A*dleman ein Public-Key-Kryptoverfahren, das nach ihnen benannte *RSA*, das auf dem *Satz von Euler* beruht. Diejenigen, die Mathematik überhaupt nicht leiden können, nehmen bitte einfach nur zur Kenntnis, dass es Euler gegeben hat. Die anderen können weiterlesen...

Für eine natürliche Zahl n definieren wir als $\phi(n)$ die Anzahl der zu n teilerfremden natürlichen Zahlen zwischen 1 und n (einschließlich). Die Funktion ϕ heißt *Euler-Funktion*. Anders ausgedrückt gibt $\phi(n)$ an, wieviele der Zahlen k zwischen 1 und n mit n den größten gemeinsamen Teiler 1 haben: ggT(k,n)=1.

An konkreten Beispielen: $\phi(1)=1$, $\phi(2)=1$, $\phi(3)=2$, $\phi(4)=2$, $\phi(5)=4$, $\phi(6)=2$, $\phi(10)=4$, $\phi(31)=30$.

$\phi(10)$ ergibt sich beispielsweise so: mit 10 haben die folgenden Zahlen zwischen 1 und 10 den größten gemeinsamen Teiler 1: 1, 3, 7, 9, also vier Stück.

Generell gilt:

1. Für eine Primzahl[98] p gilt stets: $\phi(p)=p-1$.
2. Sind p und q zwei verschiedene Primzahlen, so ist $\phi(pq)=(p-1)(q-1)$.

Die erste Aussage ist trivial; die zweite Aussage folgt aus der Überlegung, dass zwischen 1 und pq gerade die Vielfachen von p und die Vielfachen von q nicht teilerfremd[99] zu pq sind. Dabei handelt es sich also um die Zahlen p, 2p, 3p, ..., (q-1)p und um q, 2q, ..., (p-1)q sowie natürlich um pq selbst. Also sind zwischen 1 und pq teilerfremd zu pq:

$$pq - 1 - (q-1) - (p-1) = pq - q - p + 1 = (p-1)(q-1).$$

Damit kann der *Satz von Euler* formuliert werden:

Sind m und n zwei natürliche Zahlen, die teilerfremd sind, so gilt:

$$m^{\phi(n)} \bmod n = 1.$$

Einen Beweis wollen wir an dieser Stelle nicht geben; dieser findet sich in jedem besseren Buch zur Zahlentheorie (vgl. etwa [NivZuc]).

Weiterhin ist für uns die sogenannte *modulare Inverse* von Bedeutung. Sind a und b ganze Zahlen und d deren ggT, so gibt es Faktoren x und y mit $d = xa + yb$.

Sind a und b teilerfremde Zahlen, das heisst: ist der ggT von a und b gleich 1, dann exisitiert eine ganze Zahl c mit der Eigenschaft $bc \bmod a = 1$. Das bedeutet, bc lässt bei Teilen durch a den Rest 1. Die mathematische Formulierung hierfür ist: b ist *modulo* a *invertierbar*.

5.4.3. RSA: Der Algorithmus

Wenden wir uns nun der eigentlichen Frage, der RSA-Verschlüsselung, zu. Für jeden Teilnehmer sind ein öffentlicher und ein privater Schlüssel festzulegen. Hierzu werden zwei „große" Primzahlen p und q bestimmt und deren Produkt $pq = n$ berechnet. Nach dem obigen ist $\phi(n)=\phi(pq)=(p-1)(q-1)$. Schließlich werden zwei Zahlen e und d errechnet, für die die Beziehung

(5.4.3.1) $e \cdot d \bmod \phi(n) = 1$

gilt.

Die beiden Zahlen e und n werden als öffentlicher Schlüssel weitergegeben, d ist der private Schlüssel[100]. Dabei steht e auch für „Exponent", n ist der Modul. Eine Nachricht, die verschlüsselt werden soll, besteht in der Praxis aus einer Sequenz von Zeichen eines bestimmten Codes. Meist wird dies heute eine Form des ASCII sein, das spielt für die grundsätzliche Verschlüsselung aber keine Rolle.

Nehmen wir für das Prinzip einfach an, wir würden nur eine Zahl m verschlüsseln wollen, wobei m kleiner als n sei.

---

[98] Zur Erinnerung: Eine Primzahl ist eine Zahl p, die nur durch sich selbst und durch 1 ohne Rest teilbar ist. 1 selbst wird per definitionem nicht als Primzahl betrachtet.

[99] Zwei ganze Zahlen m und n heißen *teilerfremd*, wenn der größte gemeinsame Teiler von m und n gleich 1 ist. Anders formuliert: keine ganze Zahl größer als 1 teilt m und n gleichzeitig (ohne Rest).

[100] Wir erinnern an die Abkürzungen: e stand für encrypt, verschlüsseln, d für decrypt, entschlüsseln.

Dann wird die Zahl m verschlüsselt durch die folgende Formel:

(5.4.3.2) $c := m^e \bmod n$

Wendet man auf diese chiffrierte Zahl (oder allgemein: Nachricht) mittels des geheimen Schlüssels d die nachstehende Formel an, so erhält man die ursprüngliche Zahl m wieder[101].

(5.4.3.3) $m = c^d \bmod n$

Beispiel: Um die ganze Mathematik etwas greifbarer zu machen spielen wir den Mechanismus einmal mit konkreten Zahlen durch, die natürlich zu Demonstrationszwecken absichtlich (relativ) klein gewählt wurden.
Seien p=3 und q=5, dann sind n = pq = 15 und $\phi(n) = (3-1)(5-1) = 8$. Nun müssen wir e und d bestimmen, so dass die Beziehung (5.4.3.1) gilt. Einfache Werte sind z.B. e = 3 und d = 11, denn 33 mod 8 = 1.
Soll nun die Zahl m = 13 verschlüsselt werden, dann geschieht dies mit der Formel (5.4.3.2), wir erhalten so: $c = 13^3 \bmod 15 = 2197 \bmod 15 = 7$. Und das Dechiffrieren mittels (5.4.3.3) ergibt erwartungsgemäß wieder m: $7^{11} \bmod 15 = 1977326743 \bmod 15 = 13$.

In der Praxis muss ein geeigneter Modul n „sehr groß” sein, d.h. nach gängiger Handhabung derzeit mindestens 150 bis 200 (Dezimal-)Stellen besitzen. Den geheimen Schlüssel d zu knacken bedeutet mathematisch, die Zahl n in ihre Primfaktoren p und q zu zerlegen bzw., was dazu äquivalent ist, $\phi(n)$ zu bestimmen[102].

Aufgrund des sehr hohen Rechenbedarfs wird der RSA-Algorithmus heutzutage vor allem für Signaturen[103] (Unterschriften) und für einen sicheren Austausch von Schlüsseln verwendet.

5.5. PGP - Pretty Good Privacy

PGP steht für *Pretty Good Privacy* und ist ein Verschlüsselungsprogramm. Mit PGP lassen sich beliebige Texte - insbesondere natürlich auch elektronische Mails - verschlüsseln. Es wurde 1990 von Philipp Zimmermann in den USA entwickelt. Er wurde daraufhin wegen Verstoß gegen das US-amerikanische Waffenexportgesetz verfolgt! Das Vergehen Zimmermanns bestand darin, dass er PGP über Newsgruppen des Usenets verbreitet hatte. Die Anklage wurde jedoch schlussendlich fallengelassen.

Mit PGP können zum einen Texte chiffriert werden (so dass sie nur der richtige Empfänger lesen kann), andererseits kann man damit Texte signieren, d.h. unterschreiben (so dass sich jeder vergewissern kann, dass der Text wirklich von dem angegebenen Absender kommt und nicht verändert wurde).

PGP benutzt dazu u.a. die Verschlüsselungsverfahren IDEA und RSA.

IDEA (*International Data Encryption Algorithm*) ist ein symmetrisches Verfahren, das PGP IDEA zur Verschlüsselung der Botschaft verwendet.

---

[101] Den mathematischen Beweis, dass $m = m^{ed} \bmod n$ gilt, können Sie in [Beutelspacher], S. 131f, nachlesen. Wenn Sie wollen.

[102] Als anschauliches Beispiel: 1994 wurde eine 129-stellige Zahl unter erheblichem Aufwand faktorisiert.

[103] Hierzu muss lediglich ein sogenannter Hashcode für die betreffende elektronische Mail errechnet und verschlüsselt werden, was die Performance nicht in dem Maße belastet, wie es die Verschlüsselung der gesamten Mail mit sich brächte.

„IDEA arbeitet mit einer konstanten Schlüssellänge von 128 Bit und verschlüsselt die Botschaft in Blöcken von je 64 Bit. Der Hauptschlüssel wird zuerst in acht Unterschlüssel zu je 16 Bit geteilt, dann wird er um 25 Stellen nach links gerückt und wieder in acht Unterschlüssel geteilt. Dies wird sooft wiederholt, bis insgesamt 52 Unterschlüssel generiert wurden.

Dann wird der erste 64 Bit-Block in vier Teile zu je 16 Bit zerlegt. Aus diesen vier Teilen werden mit Hilfe der ersten sechs Unterschlüssel in 14 Rechenschritten vier neue Textblöcke errechnet. Dieses Verfahren wird insgesamt acht mal durchgeführt, wobei in jedem Durchgang mit den vier Ergebnis-Textblöcken des vorigen Durchgangs begonnen wird und die jeweils nächsten sechs Unterschlüssel zur Verschlüsselung verwendet werden. Mit den vier letzten Unterschlüsseln (von 52 wurden erst 48 verwendet) werden die letzten vier Ergebnis-Textblöcke ein weiteres Mal verschlüsselt"[104].

Dies geschieht für alle 64 Bit-Blöcke der Reihe nach; schließlich ergeben alle Ergebnisse zusammen die chiffrierte Nachricht. Die Decodierung findet entsprechend in umgekehrter Reihenfolge mit den entsprechend passenden „Umkehrschlüsseln" statt.

Einen sehr informativen Beitrag zu PGP finden Sie auch auf den Webseiten der Technischen Universität Chemnitz[105].

5.6. Anmerkungen zur Anonymität

Bei Kryptographie dreht es sich um Verschlüsselung, Verheimlichung. Üblicherweise wird eine Nachricht chiffriert, das heißt: der Inhalt der Nachricht wird für Dritte (hoffentlich) unlesbar gemacht. Dagegen bleibt bei der Verschlüsselung einer eMail nachvollziehbar, dass von Sender A an Empfänger B überhaupt eine Nachricht gesandt wurde.

Anonymität betrifft nun die, so betrachtet, noch ausstehenden Aspekte: das Geheimhalten des Senders und/oder Empfängers bzw. der Nachrichtenübermittlung als solcher.

Bei politischen, geheimen Wahlen ist derjenige, der seine Stimme abgibt, also der Sender der Botschaft, geheim. Ebenso verhält es sich mit Bargeld: die konkrete Münze oder der spezielle Geldschein haben kein „Protokoll" darüber, wer sie wann in seinen Händen hatte. Und auch in den mehr oder weniger beliebten Chats der Online-Dienste oder im Internet genießen die Teilnehmer ihre Anonymität[106].

Wie erreichen wir nun Anonymität des Empfängers? Ganz einfach: wenn wir „allen" (oder zumindest „vielen") die betreffende (verschlüsselte) Nachricht senden, dann kann nur der „passende" Empfänger decodieren, das heißt die betreffende Botschaft verstehen.

Aber auch das Verfahren, das bei Briefwahlen stattfindet, führt zu einer Methode, Anonymität des Empfängers zu erreichen. Hierzu stelle man sich vor, dass ein Brief in einem

---

[104] Zitiert nach Thomas [Landauer], http://www.unet.univie.ac.at/~a9204810/PGP.htm.

[105] Die URL der Seite des Autors Holger [Trapp] ist http://www.tu-chemnitz.de/~hot/pgp.html. Damit die Leserinnen und Leser des vorliegenden Textes nun nicht gleich in das Internet gehen müssen, haben wir im Anhang einen Auszug dieser Webseite wiedergegeben (siehe Abschnitt A.3 auf Seite 240).

[106] Dabei ist diese Anonymität „relativ": im Rahmen des Online-Dienstes kann durchaus bekannt sein, über welche Telefonleitung beispielsweise sich ein Teilnehmer eingewählt hat. Und auch im Internet-Chat hinterlassen die beteiligten Rechner üblicherweise ihre digitalen Spuren.

zusätzlichen Umschlag an eine Weiterverteilungsadresse geschickt wird, wo dann der äußere Umschlag entfernt wird. Und erst auf dem inneren Briefkuvert steht dann die eigentliche Zieladresse; dafür muss hier selbstverständlich (auch) keine Absenderangabe gemacht worden sein.

Schließlich bleibt zu hinterfragen, wie Dritten gegenüber die Tatsache verheimlicht werden könnte, dass überhaupt eine Nachricht übertragen wurde. Dies kann dadurch erreicht werden, dass „immer" oder zumindest „sehr oft" irgendwelche (Dummy-)Nachrichten gesendet werden. Dann fällt die eine „echte" überhaupt nicht mehr auf, das eventuelle Abhören der Datenübermittlungen und die damit verbundenen Entschlüsselungsversuche müssen fast zwangsläufig scheitern.

Am Rande sei hier auf das relativ neue Gebiet der Steganographie hingewiesen. Dabei handelt es sich um die Technik, Nachrichten „unsichtbar" in (beispielsweise) Graphiken einzubetten. Diese Methode hat in den letzten Jahren erheblichen Zuspruch gefunden. Aus Platzgründen können und wollen wir an dieser Stelle hierauf nicht weiter eingehen[107].

---

[107] Interessierte finden im Internet zahlreiche Quellen zur Steganographie, u.a. Sei die Website http://www.steganography.com/ erwähnt. Dort ist unter http://www.steganography.com/english/steganos/sd_dl.htm ein Tutorial zur Steganographie zu finden.

6. THEORETISCHE INFORMATIK UND COMPILERBAU

Das Themengebiet Theoretische Informatik umfasst an der Universität die Teile Compilerbau, Formale Sprachen, Automatentheorie, Berechenbarkeit und Entscheidbarkeit, Algorithmen und Datenstrukturen sowie Künstliche Intelligenz. Wir wollen uns hier aus Platzgründen nur mit einigen Aspekten beschäftigen[108].

6.1. Einführung in Aspekte der Theoretischen Informatik

6.1.1. Entscheidbarkeit und Berechenbarkeit (Einblick)

Ein fundamentaler Begriff der Informatik ist der des *Algorithmus*. Für viele Problemstellungen existieren Algorithmen; die folgerichtig spannende Frage ist: gibt es Problemstellungen, für die es keinen Lösungsalgorithmus gibt?

Lange Zeit glaubte man, dass man ein Problem lediglich präzise genug formulieren müsste, damit es einen Algorithmus zu dessen Lösung auch tatsächlich gibt oder man wenigstens beweisen kann, dass gar keine Lösung existiert.

Einer der bekannten Befürworter dieses Glaubens war der Mathematiker David Hilbert: "*Ein bestimmtes mathematisches Problem muss notwendigerweise einer exakten Lösung zugänglich sein, entweder in Form einer direkten Antwort auf eine gestellte Frage, oder durch den Beweis seiner Unlösbarkeit und dem damit verbundenen notwendigen Scheitern eines jeden Versuchs.*"

Hilberts Ziel war es, ein mathematisches System aufzustellen, in dem alle Probleme exakt als Aussagen formulierbar sind, die entweder wahr oder falsch sind. Seine Vorstellung war es, einen Algorithmus zu finden, der zu einer gegebenen Aussage in einem formalen System entscheidet, ob die Aussage richtig oder falsch ist.

Die Wahrheit einer Aussage in einem formalen System zu entscheiden, ist unter dem Namen Hilbertsches Entscheidungsproblem bekannt geworden. In den Dreißiger Jahren zeigte eine Reihe von Ergebnissen, dass dieses Entscheidungsproblem nicht „*berechenbar*" ist, d. h. dass es keinen solchen Algorithmus gibt, wie Hilbert ihn sich vorstellte. Ein solches Resultat war 1931 das *Unvollständigkeitstheorem* von Kurt Gödel[109], das u. a. besagt, dass es keinen Algorithmus geben kann, der als Eingabe irgendeine (beliebige) Aussage über die natürlichen Zahlen erhält und feststellt, ob diese Aussage wahr oder falsch ist. Mit dem mathematischen Begriff der Widerspruchsfreiheit spielend lässt sich als weitere verblüffende Konsequenz des Gödelschen Unvollständigkeitssatzes formulieren: *Ist die Mathematik widerspruchsfrei, so lässt sich ihre Widerspruchsfreiheit nicht mathematisch beweisen*![110]

---

[108] Etwas ausführlicher finden sich die hier besprochenen Themen in [Baeumle2].

[109] Kurt Gödel: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme* I. Monatshefte für Mathematik und Physik 38 (1931).

[110] Mathematisch interessierte und bewanderte Leser finden ausführlichere und logisch präzise Darstellungen des Gödelschen Unvollständigkeitstheorems in dem Buch *Einführung in die mathematische Logik* [EbbFluTho].

Ein konkretes Beispiel für Nicht-Berechenbarkeit ist das sogenannte Halteproblem: Ist für ein beliebiges Programm (= einen beliebigen Algorithmus) bestimmbar, ob es eine Endlosschleife enthält? Die Antwort auf das Halteproblem ist (bzw. wäre) ein Algorithmus A, der zu einem gegebenen Programm P und seinen Eingabedaten D aussagt, ob P jemals halten wird, wenn es mit den Eingabedaten D ausgeführt wird.

Im Alltag begnügt man sich mit der Vorgabe eines Zeitlimits, innerhalb dessen ein Programm ausgeführt sein muss; benötigt es mehr Zeit als vorgegeben, so wird sein Abbruch erzwungen. Aus zwei Gründen ist dies jedoch keine Ideallösung. Zum einen verschwenden nicht innerhalb der vorgegebenen Zeit beendete Programme die gesamte, ihnen zugewiesene CPU-Zeit; zum anderen kann es natürlich auch sein, dass ein Programm just vor seiner erfolgreichen Beendigung abgebrochen wird!

Nachfolgend wird gezeigt, dass das Halteproblem nicht lösbar ist.

Annahme: Es sei A ein Algorithmus, der zu den Eingaben P (Programm) und D (Daten, die das Programm P verwendet), entscheidet, ob P(D) stoppt oder endlos läuft. Stoppt P(D), so soll A "Ok" sagen, andernfalls sagt A "Nicht Ok".

Betrachten wir nun den Algorithmus B, der zu der Eingabe P (Programm) stoppen soll, wenn A(P,P) "Nicht Ok" sagt, also wenn das Programm P auf sich selbst angewendet nicht stoppt. Entsprechend soll B(P) in eine Endlosschleife laufen, wenn A(P,P) "Ok" sagt. – Sehen wir uns an, was B(B) tut:

1.Fall: B(B) stoppt.
Das ist genau dann der Fall, wenn A(B,B) "Nicht Ok" sagt,
und das ist genau dann der Fall, wenn B(B) nicht stoppt!
Dies ist ein Widerspruch.

2.Fall: B(B) stoppt nicht, läuft also endlos.
Das ist genau dann der Fall, wenn A(B,B) "Ok" sagt,
und das ist genau dann der Fall, wenn B(B) stoppt!
Dies ist ebenfalls ein Widerspruch.

Damit ist die Annahme, es könnte solch einen Algorithmus A geben, widerlegt.

6.1.2. Übersetzerarten

Ein Compiler ist ein spezieller Übersetzer. Üblicherweise werden Übersetzer unterschieden in

a) Assembler
Übersetzung einer low-level-Sprache in Maschinencode.

b) Preprocessor
Transformation einer "Obermenge" einer high-level-Sprache in die Sprache selbst. Diese Vorgehensweise ist bei der Sprache *C* oder der *Embedded SQL*-Programmierung üblich. Ebenso sind einige frühe *C++*-Compiler lediglich Preprocessoren für die Programmiersprache *C*.

c) Compiler
Übersetzung einer high-level-Sprache in Maschinencode oder eine andere (zumeist low-level-)Sprache.

d) Interpreter
Wie der Name bereits andeutet: hier wird kein (statischer) Code erzeugt; in interaktiver Kommunikation des Programmierers mit dem Interpreter wird der Quelltext Schritt für Schritt ausgeführt ("interpretiert"). Dieses Prinzip findet man vor allem bei der Programmiersprache BASIC sehr häufig (auf dem PC z. B. in der Form von GWBASIC oder QBASIC), aber auch die objektorientierte Sprache Smalltalk wird auf der Basis eines Interpreters angeboten.

Neben den verschiedenen Übersetzertypen können weitergehende Varianten von Übersetzungs-Systemen von Interesse sein. Dies sind im wesentlichen

a) Syntaxgesteuerte Editoren
Das Programm wird mit Hilfe eines von der Programmiersprache abhängigen Editors eingegeben, der z. B. menügesteuert die Auswahl aus den verschiedenen Statement-Arten anbietet und Programmskelette zur Verfügung stellt.

b) Inkrementelle Compiler
Eingebettet in eine Dialogumgebung ermöglicht der inkrementelle Compiler die Programmerstellung, Übersetzung und das Testen der Programme. Änderungen des Quelltextes verursachen keine Neuübersetzung des gesamten Programms, sondern nur der geänderten Inkremente.

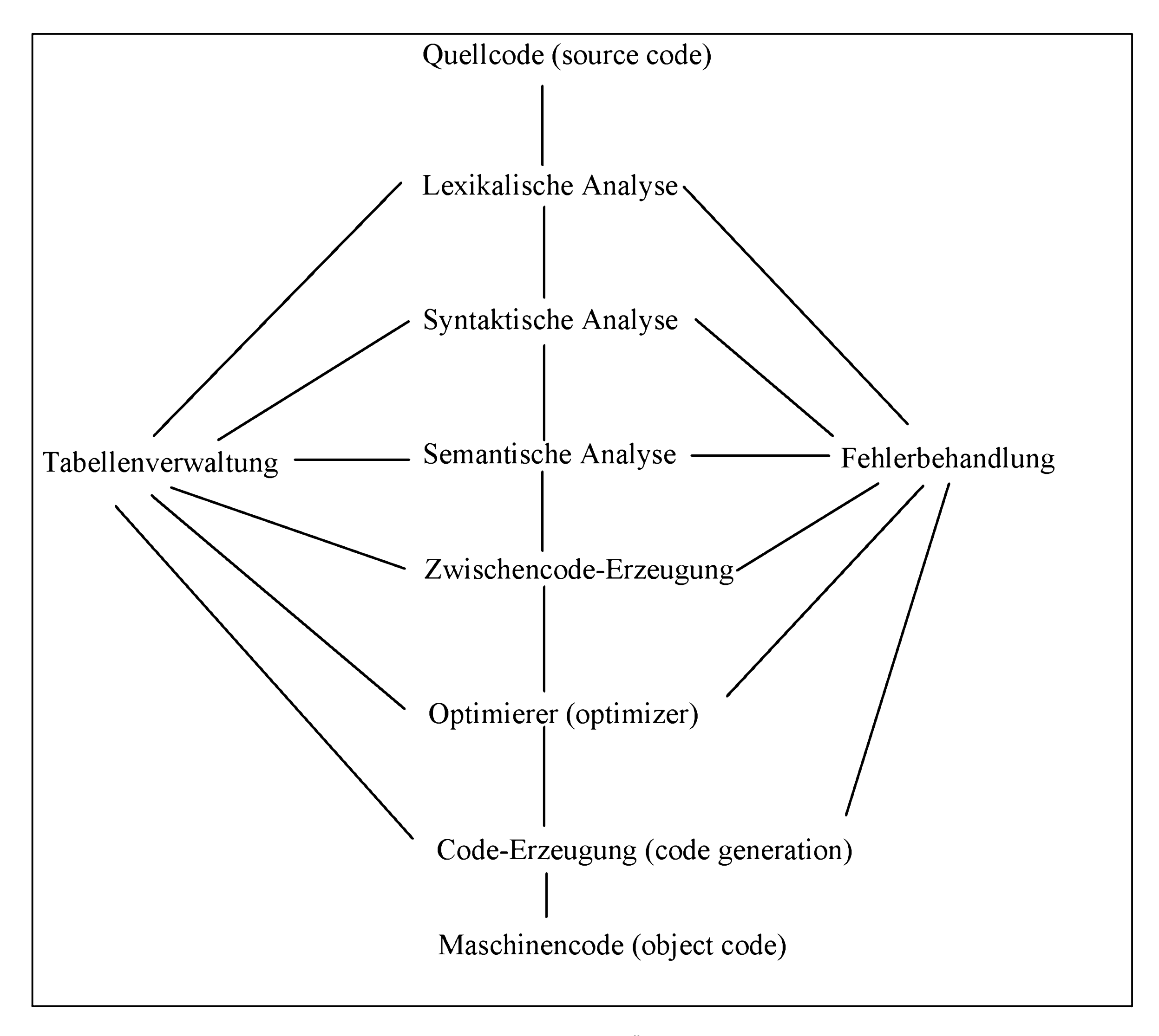
c) Disassembler, Decompiler
Rückübersetzung von Maschinencode in Assembler oder eine high-level-Sprache. (Letzteres ist i.a. nicht möglich, zumindest kann in der Regel nicht eindeutig rückübersetzt werden.)

d) Transpiler
Hier wird der Code einer high-level-Sprache in eine andere high-level-Sprache transformiert. Solche Transpiler gibt es beispielsweise für die Umwandlung von Pascal-Code in C oder von ForTran nach C.

Im Folgenden wenden wir uns schwerpunktmäßig den Arbeitsschritten eines Compilers zu.

6.1.3. Compilerphasen

Die Arbeit eines Compilers kann grob unterteilt werden in zwei Phasen(-gruppen): einmal ein Analyseteil, zum andern ein Syntheseteil; ein Compiler prüft in den Analysephasen zunächst den zu verarbeitenden Quelltext einer Pro- grammiersprache auf (lexikalische, syntaktische und semantische) Korrektheit und überführt ihn, falls diese Überprüfung zufriedenstellend verläuft, anschließend im Rahmen der Synthesephasen in eine *target-language* (Zielsprache). Diese Überführung kann eventuell auch über Zwischensprachen erfolgen. Die Zielsprache ist der sogenannte *object code*, der in der PC-Welt als .OBJ-Datei und bei UNIX als .o-Datei vorliegt.



Compilerphasen im Überblick

6.1.3.1. Lexikalische Analyse

Eingabe ist das Quellprogramm P, d. h. eine (endliche) Folge von Zeichen (Buchstaben, Ziffern, zulässige Sonderzeichen). Die Verarbeitung dieser Phase besteht darin, Grundsymbole aus den Zeichen zusammenzubauen. Beispielsweise wird das Pascal-Schlüsselwort **begin** aus den ASCII-Zeichen 'b', 'e', 'g', 'i' und 'n' zusammengesetzt, die Zahl 357 wird aus den Ziffern (digits) '3', '5' und '7' ermittelt. Dieser Vorgang wird auch als *Scannen* bezeichnet, ein Programm, das die lexikalische Analysephase realisiert, entsprechend als *Scanner*.

Üblicherweise werden bereits in dieser Phase Tabellen bereitgestellt (initialisiert), in die später Bezeichner (identifier), Konstanten (Literale) und Datentypen eingetragen werden. Als Initialisierung sind hier bereits die Standardbezeichner festgehalten.

Die Ausgabe ist ein Programm P' in Form einer Folge von Symbole. So kann etwa aus der Anweisung

```
if x > 0 then y := 1;
```

die Symbolfolge

```
ifsym ident gtr number thensym ident assign number semikolon
```

generiert werden[111]. Hierbei wird die Ebene der einzelnen ASCII-Zeichen verlassen, d. h. es spielt insbesondere keine Rolle, ob das Pascal-Schlüsselwort `if` in Groß- oder in Kleinbuchstaben geschrieben ist.

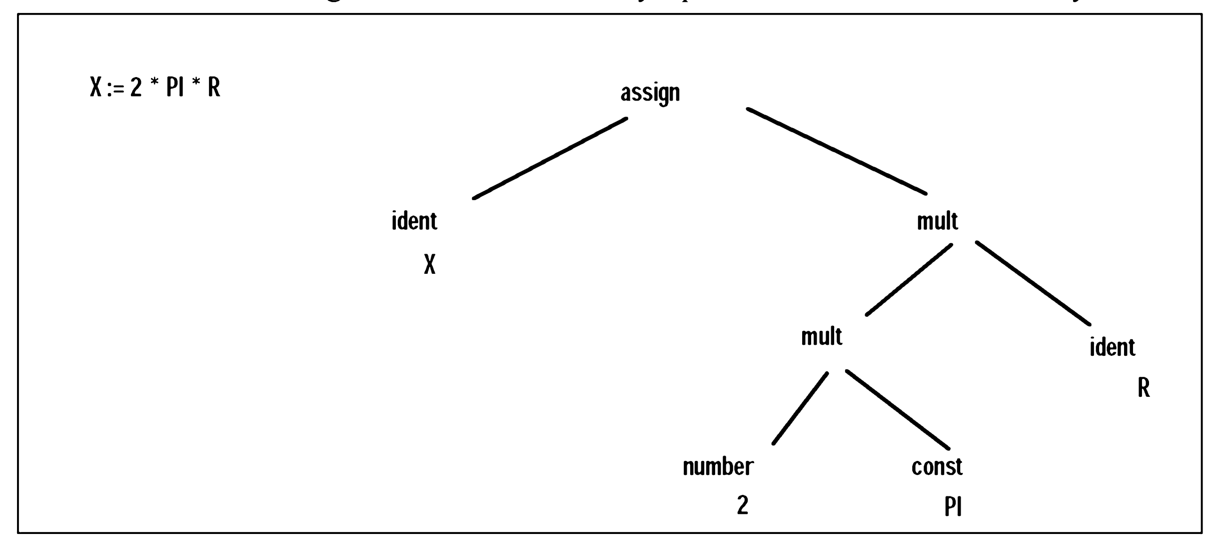
In der Literatur wird empfohlen, sich für die Entwicklung eines Compilers auf eine feste Darstellung dieser "Token" festzulegen, z. B. auf (Art, Wert)-Paare (hier etwa: (keyword, 'begin') und (number, 357)). Dann ist es irrelevant, ob der Wert direkt im Token vorliegt oder als Verweis in eine Tabelle.

---

[111] Hier werden bereits die Symbol-Bezeichnungen des von uns später behandelten Parsers verwendet. Dabei steht ifsym für das Symbol zum Schlüsselwort if, ident für Identifier, Bezeichner, gtr steht für greater, den größer-Operator, thensym steht für das Schlüsselwort then, assign für die Zuweisung (assignment), number für eine Zahl und semicolon für das Sonderzeichen ";", eben das Semikolon.

6.1.3.2. Syntaktische Analyse

Eingabe ist das Programm P', das die lexikalische Analyse erzeugt hat. Es wird hier geprüft, ob die formale Aufeinanderfolge von Symbolen, die sogenannte Syntax der entsprechenden (Programmier-)Sprache erfüllt wird. Neben den Ergänzungen der Tabellen (Deklarationsteil) wird ein sogenannter *abstrakter Baum* A erzeugt (Statement-Teil). Dieser beinhaltet wichtige Informationen für die folgende semantische Analysephase und die anschließende Synthese.



Ein abstrakter Baum

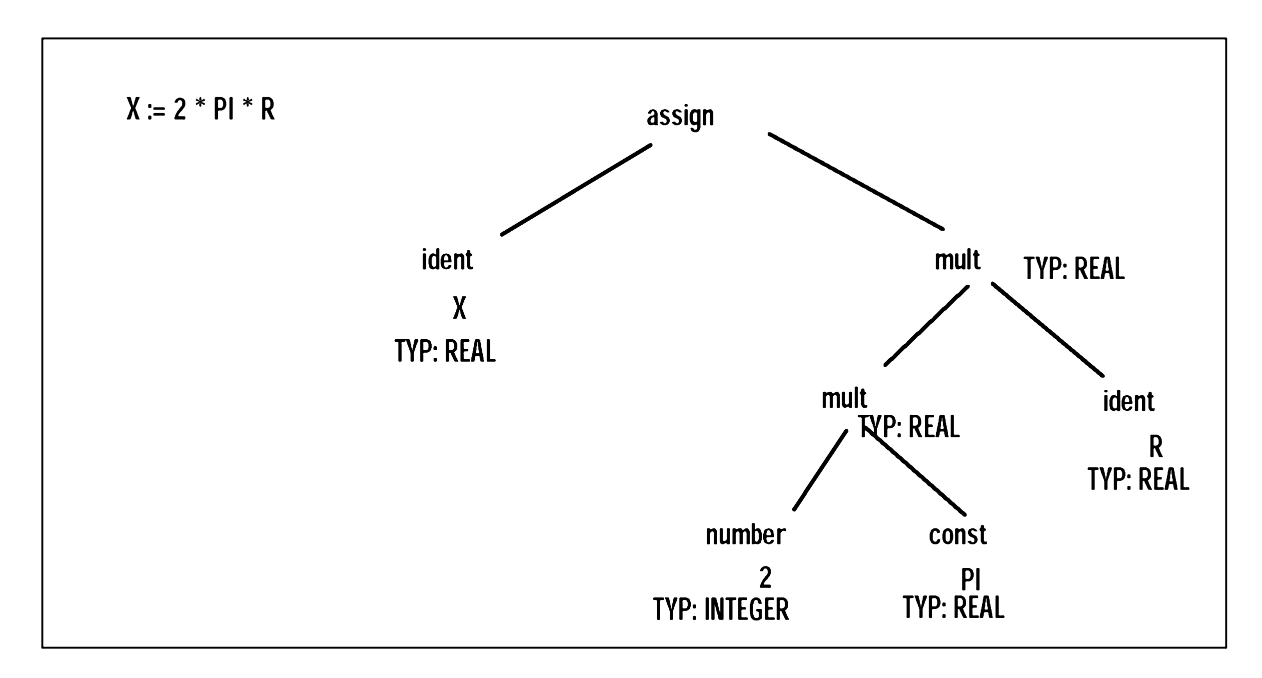
Hierbei bedeuten: ident = Identifier, Bezeichner; const = Konstante; mult = Multiplikation

6.1.3.3. Semantische Analyse

Eingabe ist der abstrakte Baum, den die syntaktische Analysephase erzeugt hat. Geprüft werden von der semantischen Analyse die Kontextbedingungen, z. B. die Typenverträglichkeit oder Typengleichheit. (Man bezeichnet dies als *statische*, zur Compilezeit überprüfbare Semantik; im Gegensatz dazu steht die *dynamische*, erst zur Laufzeit relevante Semantik.) Den Knoten des abstrakten Baumes A werden semantisch relevante Informationen zugewiesen, es entsteht ein *attributierter* abstrakter Baum A^a und es werden die zu verschiedenen Symbolen notwendigen Informationen in *Symboltabellen* eingetragen[112]. Die Ausgabe der semantischen Analyse ist der erwähnte attributierte abstrakte Baum A^a.

---

[112] Für einen Pascal-Compiler sind dies Informationen zu Bezeichnern. Bei einem Compiler für C++ muss jedoch auch ein Eintrag für einen überladenen Operator + in die Symboltabelle vorgenommen werden, da das Symbol + (*plus*) hier erst zusammen mit seinem Kontext (der Parameterliste) korrekt auszuwerten ist.



Ein attributierter abstrakter Baum

6.1.3.4. Zwischencode-Erzeugung

Eingabe der Zwischencode-Erzeugung ist der attributierte Baum A^a. Es wird ein Zwischencode P_Z erzeugt (in einer Zwischensprache Z), dessen Vorteil es vor allem sein kann, maschinenunabhängig zu sein. Die Ausgabe ist dieser Zwischencode P_Z.

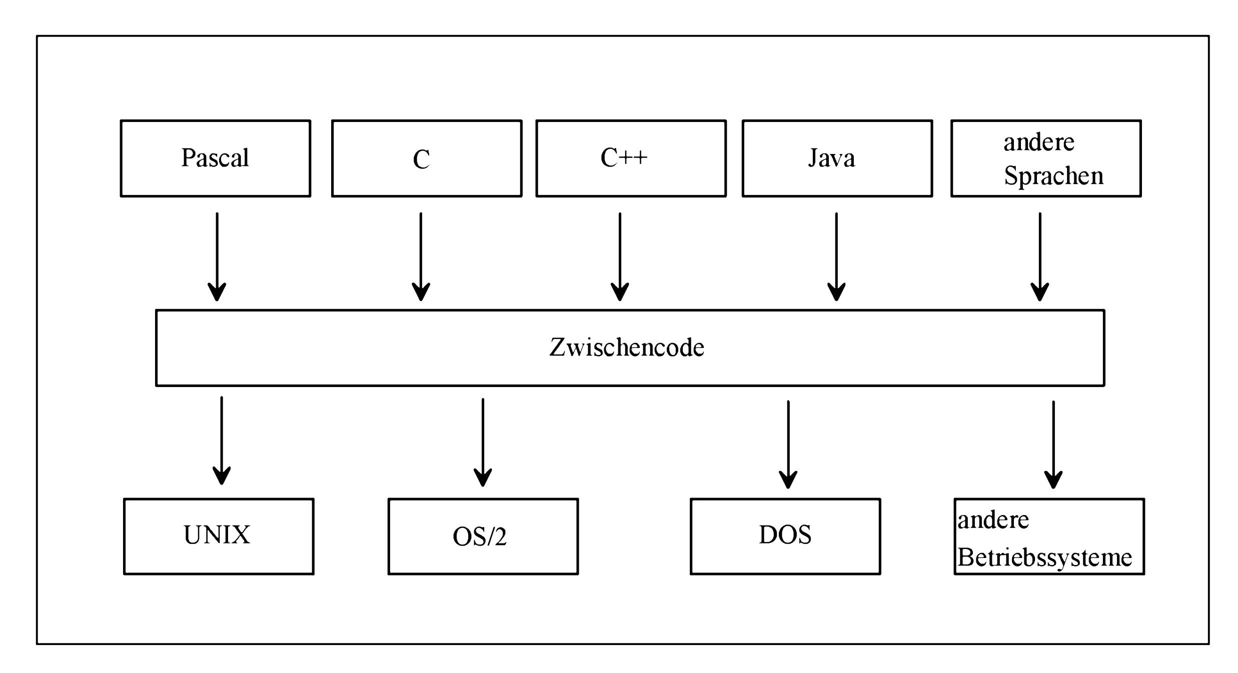
Der Zwischencode kann durchaus maschinenorientiert festgelegt werden. Zu denken wäre dabei an Zwei- oder Dreiadreßcode, je nach Komplexität. Beispiel: aus A:=B*C wird dann im ersten Fall[113] (Mult B,C; Store Erg,A) und im zweiten Fall (Mult B,C,A). Seltener wird ein aufwendiger Einadresscode verwendet; für unsere Zwecke schließlich reicht eine symbolische Notation (siehe Abb. 1.5.).

Außerdem sei darauf hingewiesen, dass ein Compilerhersteller ohne großen Aufwand Compiler für andere Programmiersprachen entwickeln kann, die denselben Zwischencode ansteuern. Um etwa einen Pascal-, einen C-, einen C++- und einen Java-Compiler für die drei Betriebssysteme UNIX, OS/2 und DOS zu entwerfen, sind gemäß nachstehender Skizze somit nur wenige Bausteine erforderlich[114]. Und andere Programmiersprachen sowie weitere Betriebssystemplattformen können leicht ergänzt werden...

---

[113] Mult steht hier wiederum als mnemonisches Kürzel für Multiplikation. "Store Erg, A" steht dabei für den Befehl, den Inhalt von A in der Variablen Erg zu speichern.

[114] Hier wird für den Moment davon abstrahiert, dass üblicherweise Java-Compiler einen sogenannten Byte-Code erzeugen, der mit dem hier vorgestellten Zwischencode nicht direkt vergleichbar ist.



Nutzung des Zwischencodes für verschiedene Sprachen und Plattformen

6.1.3.5. Optimierung

Die (optionale) Optimierungsphase nimmt den Zwischencode P_Z und eliminiert redundante (d. h. de facto wirkungslose) oder "tote" Anweisungen, optimiert Variablenzugriffe usw. Eine solche Phase ist generell auch später, nach der Maschinencode-Erzeugung, noch möglich und in der Praxis vielfach realisiert. Ausgabe ist ein optimierter Zwischencode $P_{Z,opt}$.

Bei der Optimierung wird generell unterschieden zwischen globaler und lokaler Optimierung. Dabei muss für die globale Optimierung eine Datenflußanalyse für das gesamte Programm durchgeführt werden, während bei der lokalen Optimierung nur *Basisblöcke* untersucht werden. Das sind Folgen von Anweisungen, die linear ohne Bedingungen durchlaufen werden (also von der ersten bis zur letzten Anweisung).

Typische Optimierungen in Basisblöcken:
– Konstanten erkennen und verbreiten (in andere Anweisungen übertragen);
– Konstanten falten (Operationen mit Konstanten zur Compilezeit ausführen);
– Algebraische Gesetze anwenden;
– Überflüssige Anweisungen entfernen;
– Speicherzugriffe optimieren (z. B. bei Array-Zugriff in Schleifen).

```
WHILE (1 < P) AND (P < 3) DO P := P + Q     (* Pascal *)


1)   L0:   IF 1 < P GOTO L1                 (* Zwischencode *)
           GOTO L3
     L1:   IF P < 3 GOTO L2
           GOTO L3
     L2:   P := P + Q
           GOTO L0
     L3:   CONTINUE


2)   L0:   IF 1 >= P GOTO L1                (* optimierter  *)
```

Kleines Beispiel einer Optimierung im Zwischencode

6.1.3.6. Code-Erzeugung

Der übergebene Zwischencode $P_{Z,opt}$ (oder ggf. P_Z, wenn keine Optimierung stattgefunden hat) wird nun übersetzt in den (endgültigen) Maschinencode P_M. (Natürlich ist dieser extrem hardwarenah und damit maschinenabhängig! Nur selten gibt es sogenannte Binärkompatibilität, die einen Austausch von Maschinencode verschiedener Rechnersysteme ermöglicht.)

6.1.3.7. Fehlerbehandlung

Eine Fehlerbehandlung ist grundsätzlich in allen Phasen möglich (und üblich); bei der Synthese sind dies jedoch nur noch Fehler im Wechselspiel mit den konkreten Systembedingungen. Generell werden Fehlermeldungen in eine Fehlerdatei bzw. Fehlertabelle (*error table*) geschrieben.
Ein guter Compiler sollte Korrekturversuche (bzw. -vorschläge) vornehmen sowie nach Fehlern (auch nach "schlimmen") wiederaufsetzen können. Die Analysephasen sollten auf jeden Fall für das gesamte Programm durchgeführt werden, damit die Benutzer Hinweise zur Korrektur erhalten.

Die möglichen (lexikalischen und syntaktischen) Fehler können verschiedenster Natur sein:

`sin(x`	fehlendes Zeichen (bzw. Symbol)
`if A then B; else C`	"überflüssiges" Zeichen
`funktion cosinus`	falsches Zeichen
`fnuction ...`	vertauschte (verdrehte) Zeichen

Probleme bei der Zuordnung eines Fehlers treten selbst in sehr einfachen Zusammenhängen auf. Beispielsweise ist bei `A := B * C - D + E )` nicht ersichtlich, ob eine

schließende Klammer zuviel oder eine öffnende zuwenig geschrieben wurde. Und im letzteren Fall: wohin gehört sie?

Einen theoretischen Ansatz einer Fehlerbehandlung liefert die sogenannte Minimale Distanz-Korrektur: mit Annahmen über die jeweilige Fehlerwahrscheinlichkeit wird diejenige Korrektur vollzogen, die den geringsten Aufwand bzw. Vertauschungsgrad bedeutet. Bei Gleichheit von Vertauschungen o. ä. werden die Wahrscheinlichkeiten zur Bewertung mit herangezogen. – Da dieser Ansatz extrem aufwendig ist, wird er in der Praxis meist nicht realisiert!

Etwas simpler gestrickt ist da der Panik-Modus: ab der Stelle, an der ein Fehler auftritt (d. h. präziser: vom Compiler registriert wurde), werden alle Zeichen ignoriert, bis wieder ein Symbol analysiert werden kann.

So kann zum Beispiel bei dem Pascal-Fragment `IF (A > B > C THEN ...` die IF-Bedingung natürlich nicht korrekt analysiert werden; indes kann der Compiler bei dem Symbol (Schlüsselwort) THEN wieder aufsetzen, d. h. so weiterarbeiten, als ob alles korrekt gewesen wäre, und ordnungsgemäß einen Fehler in dem davorstehenden (mißglückten) booleschen Ausdruck melden.

6.2. Formale Sprachen

Die zuvor vorgestellten Analysephasen des Compilers wurden auf Basis der sogenannten Formalen Sprachen entwickelt.

Eine Sprache kann – wie mathematisch jede Menge – festgelegt werden durch eine komplette Liste aller gewünschten Elemente. Aber schon bei den natürlichen Sprachen wird auf diesen Aufwand verzichtet, indem Regeln zur Konstruktion aller möglichen Sätze definiert werden. Wir alle beherrschen diese Regeln mehr oder weniger gut.

Bei den Formalen Sprachen betrachten wir – im Zusammenhang mit der Aufgabe, einen Compiler für eine Programmiersprache zu schreiben – verschiedene Beschreibungsmechanismen für die Erzeugung aller "erwünschten" Zeichenfolgen einer Sprache. Wir lösen uns dabei z. T. von den umgangssprachlich geläufigen Begriffen wie Zeichen, Wort, Satz.

Beispiel: *program mini(output); begin writeln('Hello, world!') end.* ist eine aus Zeichen zusammengesetzte Zeichenfolge (ein Wort), die einen Satz der Programmiersprache Pascal darstellt.

Im nächsten Abschnitt beschäftigen wir uns zunächst mit der Spracherzeugung mittels Grammatiken.

6.2.1. Spracherzeugung, Grammatiken

Es sei eine Grundzeichenmenge Φ festgelegt, z. B. der ASCII-Code. (In der Regel ist dieser Grundzeichenvorrat mit dem Pascal-Datentyp char modellierbar.)

Nachstehend sollen die wesentlichen Begriffe für die weitere Arbeit mit Formalen Sprachen vorgestellt werden.

6.2.1.1. Definitionen (Alphabet, Wort, Länge)

1. Ein *Alphabet* A ist eine endliche Menge von Zeichen, d. h. von wohldefinierten und wohlunterscheidbaren Elementareinheiten. (Mathematisch: A ist eine Teilmenge von Φ.)
2. Ein *Wort* w über dem Alphabet A ist eine endliche Sequenz von Zeichen aus A.
3. Die *Menge aller Worte* über dem Alphabet A wird mit A^* bezeichnet.
4. Die *Länge* eines Wortes w aus A^* ist definiert als die Anzahl der Zeichen in w.
   Notation hierfür: |w|.
5. Speziell ist das Wort, das aus 0 Zeichen besteht, das sogenannte *leere Wort*, das wir mit dem Symbol ε bezeichnen wollen.
6. Eine *Sprache* L (für "*language*") ist eine Teilmenge von A^*.
7. Sind w_1 und w_2 Worte über A (also: $w_1, w_2 \in A^*$), so heißt w_1w_2 die *Konkatenation* von w_1 und w_2.

6.2.1.2. Beispiele (Alphabet, Wort)

a) Ist A := { 0, 1 }, so ist w := 110 ein Wort über A (bzw. ein Element von A^*). Die Länge von w ist |w|=3.

b) Ist A := { *a*, *b*, *c* }, so sind w_1 := *aab* und w_2 := *cc* zwei Worte über A. Die Länge von w_1 ist $|w_1| = 3$, die von w_2 ist $|w_2| = 2$. Die Konkatenation der beiden Worte führt zu w_1w_2=*aabcc*; w_2w_1 ist entsprechend *ccaab*.

c) Sei A := { *x* }, L_1 := { *x*, *xx*, *xxx*, ... }, dann ist die Sprache L_1 nicht streng formal korrekt, dafür aber intuitiv verständlich angegeben. Offenbar sind alle Worte über A mit Ausnahme des leeren Wortes ε in der Sprache L_1 enthalten.

d) Sei A := { 0,1,2,3,4,5,6,7,8,9 } und L_2 := { $w \in A^*$ | Das 1.Zeichen von w ist nicht 0; w ist nicht das leere Wort }, dann beschreibt L_2 offenbar gerade die Sprache aller positiven Dezimalzahlen ohne führende Nullen.

Nach diesen Beispielen ist es offensichtlich sinnvoll, sich Notationen zur dynamischen Her- oder Ableitung einer Sprache auszudenken. Dies führt (u. a.) zu den sogenannten Grammatiken.

6.2.1.3. Definition (Grammatik, Terminalzeichen, Nichtterminalzeichen)

Eine *Grammatik* G ist ein Viertupel (vornehm auch Quadrupel genannt) (N,T,S,P) und besteht aus einer Menge N von *Nichtterminalzeichen*, aus einer Menge T von *Terminalzeichen*, einem *Startsymbol* S ∈ N und einer Menge P von *Produktionsregeln*. Dabei

sind alle Mengen endlich, N und T haben keine gemeinsamen Elemente (d. h. sie sind disjunkt: $N \cap T = \emptyset$).

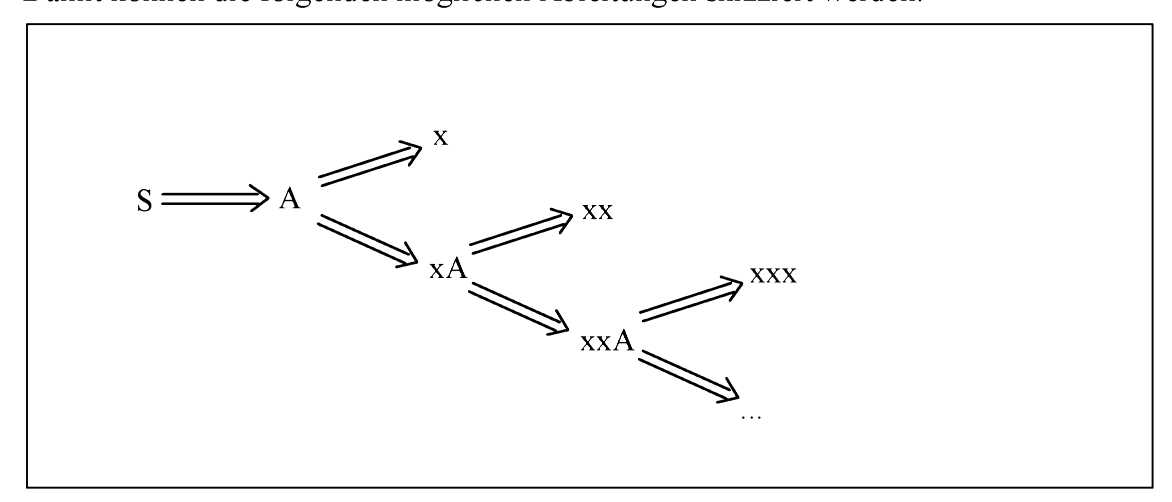
Bevor wir in Abschnitt 6.2.1.5. die formale Definition der Arbeitsweise einer Grammatik betrachten, hier eine kurze (umgangssprachliche) Beschreibung ihrer Funktion.

Eine Sprache ist bekanntlich eine Menge von Worten über einem Alphabet. Mithilfe einer Grammatik soll eine Sprache erzeugt werden. Mit den Produktionsregeln liefert die Grammatik die Vorschriften für diese Erzeugung. Dazu bedient sie sich nicht nur der Zeichen, die ein Wort enthalten kann, sondern auch noch weiterer Zeichen. Da diese Zeichen nur während des Produzierens, nicht aber mehr im erzeugten Wort auftreten, heißen sie *Nichtterminalzeichen*; die Elemente des Sprachalphabets heißen analog *Terminalzeichen*.

Die Nichtterminalzeichen dienen gewissermaßen als das Werkzeug, damit letztlich eine Folge, nur aus Terminalzeichen bestehend, gemäß den in P genannten Produktionsregeln erzeugt werden kann: ausgehend vom Startsymbol werden sukzessive Produktionsregeln angewendet, bis sämtliche Nichtterminalzeichen ersetzt worden sind. Eine solche Produktionsregel wird dabei notiert in der Form A ::= B, lies: *A wird abgeleitet zu B*.

6.2.1.4. Beispiele (Sprache, Terminalzeichen, Nichtterminalzeichen)

a) Zur Sprache L_1 aus Beispiel 6.2.1.2.c):
   Mit der Nichtterminalzeichenmenge N := { S, A } und der Terminalzeichenmenge T := { *x* } können Produktionsregeln P angegeben werden, die zur Erzeugung der Sprache L_1 dienen:
   S ::= A, A ::= *x*, A ::= *x*A.
   Damit können die folgenden möglichen Ableitungen skizziert werden.



   Auf diese Weise entsteht (jeweils in endlich vielen Schritten) jeder mögliche Satz der Sprache L_1.
   Der Begriff Erzeugung wird nachfolgend noch präzise definiert!

b) Sei L_{dual} die Sprache aller Dualzahlen (0, 1, 10, 11 usw.). Dann erzeugt die nachfolgend genannte Grammatik G := (N,T,S,P) die Sprache L_{dual}.
   N := { S, A }

T := { 0, 1 }

P := { S ::= 0; S ::= 1A; A ::= 0A; A ::= 1A; A ::= ε }

6.2.1.5. Definitionen (Ableitung u.a.)

1. Als *Grammatik-Alphabet* wird die Vereinigungsmenge $N \cup T$ von N mit T bezeichnet.

2. Sei G eine Grammatik, G := { N, T, S, P }; seien v und w Worte über dem Grammatik-Alphabet. Dann sagen wir: w lässt sich *direkt ableiten* aus v (oder: w ist eine *direkte Ableitung* von v), wenn gilt:
   $v = xUy$, $w = xzy$, $x,y,z \in (N \cup T)^*$, $U \in (N \cup T)^*N(N \cup T)^*$,
   $U ::= z \in P$ (ist eine Produktionsregel)
   Für diesen Sachverhalt schreiben wir $v \Rightarrow w$.

3. v *erzeugt* w (bzw. w *ist ableitbar aus* v), falls es eine natürliche Zahl $n > 0$ und $u_0,\ldots,u_n \in (N \cup T)^*$ gibt mit $v=u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n=w$. Notation: $v \,{}^+\!\!\Rightarrow w$.

4. Eine Grammatik G heißt *zyklenfrei*, wenn für kein $X \in N$ gilt: $X \,{}^+\!\!\Rightarrow X$, d. h. wenn es keine Zyklen gibt, bei denen von X ausgegangen wieder X erzeugt werden kann. (Dieser Begriff dient faktisch nur dazu, sich auf "sinnvolle" Produktionsregeln zu beschränken.)

5. Eine Sequenz $u_0,\ldots,u_n$ mit $u_0,\ldots,u_n \in (N \cup T)^*$ und $u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n$ heißt *Ableitung der Länge n*.

6. Gilt $S \,{}^+\!\!\Rightarrow x$ für ein $x \in (N \cup T)^*$, so heißt x eine *Satzform* der Grammatik G.

7. Ein *Satz* ist eine Satzform, die nur aus Terminalzeichen besteht, d. h. mathematisch formuliert: x ist ein Satz $\Leftrightarrow$ ($S \,{}^+\!\!\Rightarrow x$, $x \in T^*$).

8. Die durch die Grammatik G *erzeugte Sprache* ist die Menge aller Sätze und heißt $L(G) := \{ x \in T^* \mid S \,{}^+\!\!\Rightarrow x \}$.

9. Zwei Grammatiken G_1 und G_2 heißen *äquivalent*, falls sie dieselbe Sprache erzeugen, d. h. wenn gilt $L(G_1)=L(G_2)$ (Symbol: $G_1 \cong G_2$).

6.2.1.6. Beispiel (Grammatik, Ableitung)

Sei G := { N, T, S, P } mit N := { S, A, B }, T := { *a*, *b* } und P definiert als die nachfolgende Menge von Produktionsregeln.

{ S ::= AB; A ::= *a*; A ::= *a*A; B ::= *bb*; B ::= BB }.

Dann sind beispielsweise *baba*, AA*a* oder *aabb* Worte über dem Grammatik-Alphabet. *a*A*bb* ist ein Beispiel für eine Satzform und *aaabbbb* ist ein Beispiel für einen Satz der von G erzeugten Sprache L(G).

Aufgabe für Sie: Versuchen Sie bitte, L(G) verbal vollständig zu beschreiben.

6.2.1.7. Die Backus-Naur-Form

Da es mitunter sehr mühselig ist, eine Sequenz von Ableitungs- oder Produktionsregeln wie z. B. A::=B, A::=C, A::=D, A::=E (usw.) anzugeben, gibt es eine Konvention, sich die Schreibarbeit etwas zu vereinfachen, die sogenannte *Backus-Naur-Form* (*BNF*). Die BNF findet sich z. B. bei Pascal, wenn gesagt wird, dass ein vollständiges Pascal-Programm aus dem Programmkopf und dem Programmblock besteht. In BNF kann diese Aussage so aussehen:

<program> ::= <program-head> <program-block>.

Das heißt: häufig werden die Nichtterminalzeichen durch in spitzen Klammern stehende selbstsprechende Abkürzungen dargestellt. Für die Notation der Produktionsregeln gelten folgende Abkürzungen.

Mit "|" wird eine Alternative bezeichnet.

A ::= B | C | D steht für A ::= B, A ::= C, A ::= D .

Mit "{ ... }" wird 0- oder mehrmalige Wiederholung dargestellt.

S ::= { xxx } steht somit für S ::= ε, S ::= xxxS .

Mit "[...]" wird eine Option (0 oder einmal) gekennzeichnet.

S ::= [A] steht für S ::= ε, S ::= A .

Daneben können die gewöhnlichen runden Klammern verwendet werden, um eine Zusammenfassung auszudrücken. Oft werden auch zur Verdeutlichung die Terminalzeichen in doppelte Hochkommata gesetzt.

6.2.1.8. Beispiel zur Backus-Naur-Form

Eine Beispielgrammatik für ganz simple arithmetische Ausdrücke könnte in BNF folgendermaßen angegeben werden.

```
N := { S, <Ausdruck>, <Term>, <Faktor>, <Bezeichner> }
T := { "+", "*", "(", ")", "A", "B", ... , "Z" }
P := { S              ::=    <Ausdruck> ,
       <Ausdruck>     ::=    <Term> [ "+" <Ausdruck> ] ,
       <Term>         ::=    <Faktor> [ "*" <Term> ]       ,
       <Faktor>       ::=    <Bezeichner> | "(" [ <Ausdruck> ] ")" ,
       <Bezeichner> ::=  "A" | "B" | "C" | "D" | "E" | "F" |
                             "G" | "H" | "I" | "J" |  "K" | "L" |
                             "M" | "N" | "O" | "P" | "Q" | "R" |
                             "S" | "T" | "U" | "V" | "W" | "X" |
                             "Y"  | "Z"
     }
```

6.2.1.9. Beispiel (COPY-Kommando)

Nachfolgend sehen Sie die (seit MS-DOS Version 5.0 verfügbare) Online-Hilfe des *COPY*-Kommandos. Die darin verwendeten eckigen Klammern sowie der senkrechte Strich stimmen in ihrer Verwendung mit der formalen Syntax der Backus-Naur-Form überein. Statt der intuitiven "..."-Schreibweise werden in BNF jedoch die geschweiften Klammern verwendet.

```
C:\>copy /?

Kopiert eine oder mehrere Dateien an eine andere Position.


COPY [/A | /B] Quelle [/A | /B] [+ Quelle [/A | /B] [+ ...]]
     [Ziel [/A | /B]]  [/V]


 Quelle   Bezeichnet die zu kopierende(n) Datei(en).

 /A       Weist auf eine ASCII-Textdatei hin.

 /B       Weist auf eine Binärdatei hin.

 Ziel     Bezeichnet Verzeichnis und/oder Dateiname der neuen
          Datei(en).

 /V       Überprüft, dass die neuen Dateien richtig
          aufgezeichnet wurden.


Um Dateien aneinanderzuhängen, geben Sie eine einzelne Datei als Ziel
an, aber mehrere Dateien als Quelle (unter Verwendung von Platzhaltern
oder in der Form: Datei1 + Datei2 + ...).
```

Die obige Syntax-Zeile sieht in Backus-Naur-Form so aus:
COPY [/A | /B] source [/A | /B] { + source [/A | /B] } [destination [/A | /B]] [/V] .

6.2.1.10. Syntaxgraphen (Syntaxdiagramme)

Eine andere übliche Darstellungsform von Grammatiken (neben der BNF) sind *Syntaxdiagramme* oder *Syntaxgraphen*.

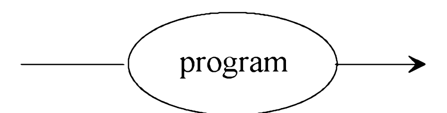
Syntaxdiagramme bestehen aus den folgenden Komponenten:

Eckige Kästchen:



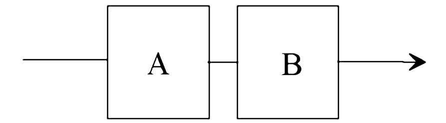
Hiermit wird auf ein anderes Syntaxdiagramm verwiesen, das den Namen "Bezeichner" trägt.

Runde (ovale) Symbole:



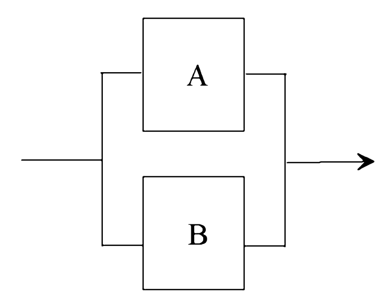
Hiermit wird ausgedrückt, dass an der entsprechenden Stelle genau das einzusetzen ist, was in dem runden Symbol steht, z. B. das genannte Schlüsselwort (hier "*program*").

Konkatenation (Aufeinanderfolge):



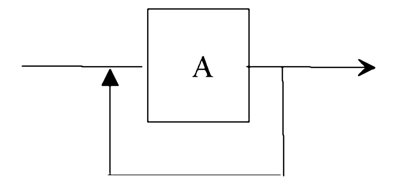
Dies steht für das, was das Syntaxdiagramm A ausdrückt, gefolgt von dem, was B beinhaltet.

Alternative (Verzweigung):



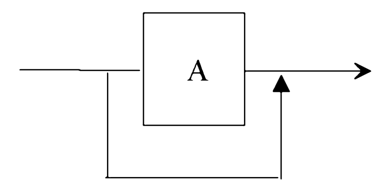
Dies steht für (wahlweise) eines der Syntaxdiagramme A oder B. (Entsprechend ist diese Diagrammform natürlich auch für eine mehrfache Alternative verwendbar.)

Wiederholungsstruktur:



Dies steht für (nur) A oder die ein- oder mehrmalige Wiederholung von A.

Option (Spezialfall der Alternative):



Dies steht für "A oder nichts".

Anmerkung: All diese Elementarbausteine sind natürlich ineinander verschachtelbar.

Das Beispiel 6.2.1.8. sieht in Form von Syntaxdiagrammen aus wie folgt:

Ausdruck

Term

+

Ausdruck

Term

Faktor

*

Term

Faktor

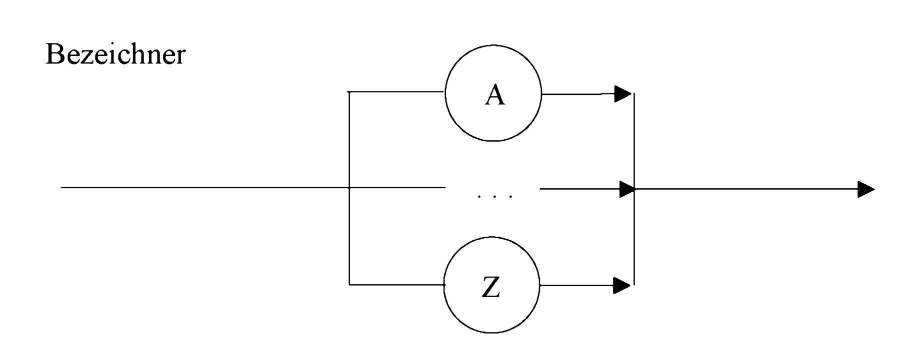
Bezeichner

(

)

Ausdruck

Bezeichner



6.3. Automatentheorie

Nachdem wir uns im letzten Abschnitt mit der Erzeugung von Sprachen verschiedener Typen beschäftigt haben, wollen wir uns nun dem Themengebiet der Spracherkennung widmen. Dabei wird nicht mehr gefragt, welche Zeichenfolgen erzeugt werden können, sondern vielmehr danach, ob eine gegebene Zeichenfolge zu einer Sprache gehört, also korrekt ist.

Ein (Beschreibungs-)Mittel für die Grundsymbole einer Programmiersprache ist das Modell des Endlichen Automaten.

6.3.1. Deterministische Endliche Automaten

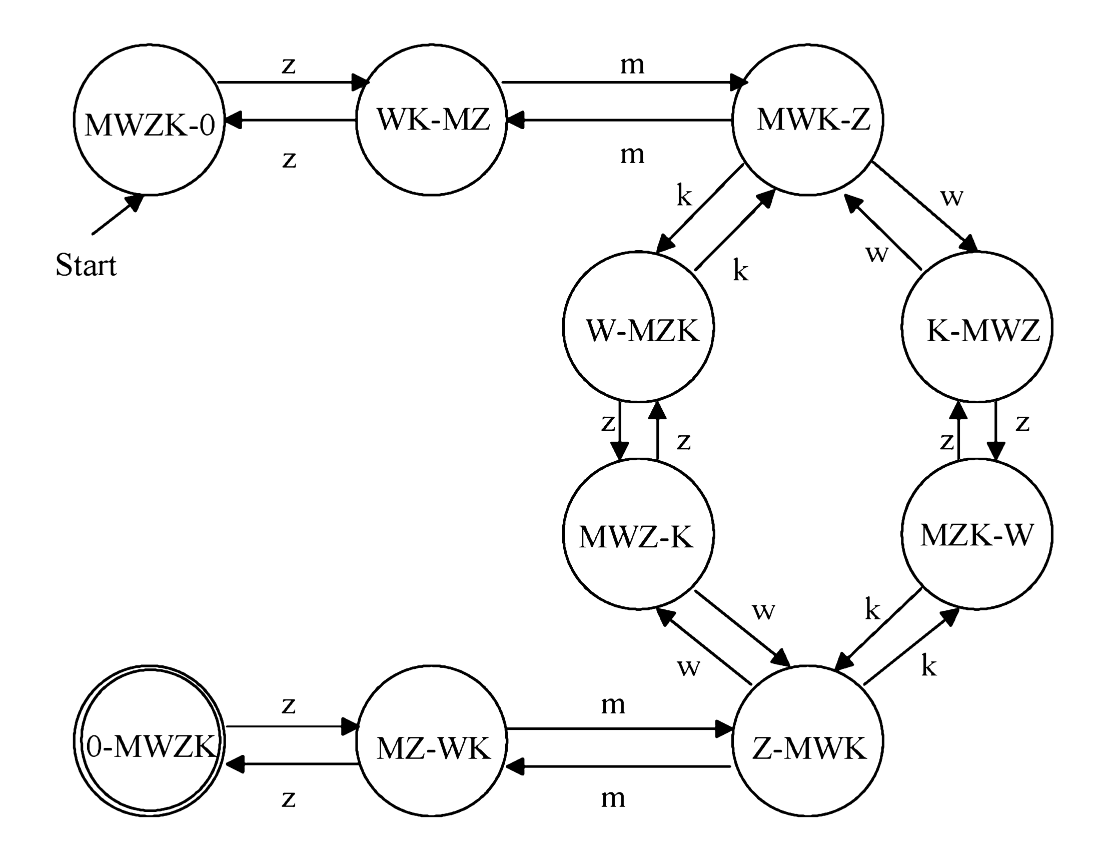
6.3.1.1. Beispiel

Zum Einstieg wollen wir anhand eines einfachen Beispiels die Aufgabe der Spracherkennung verdeutlichen.

Vielleicht kennen Sie das Knobelspiel, bei dem ein Mensch die Aufgabe hat, mittels eines Bootes einen Wolf, eine Ziege und einen Kohlkopf von der einen auf die andere Seite eines Flusses überzusetzen. Nebenbedingungen: Er darf immer nur alleine oder mit einem der drei im Boot übersetzen und außerdem dürfen nie Wolf und Ziege oder Ziege und Kohl alleine am Ufer zurückbleiben, denn sie haben sich zum Fressen gern.

Hier geht es bei der Spracherkennung darum, eine vorgeschlagene Lösung zu prüfen. Schlägt etwa jemand vor, zunächst solle der Mensch mit der Ziege übersetzen, dann alleine zurück, dann mit dem Wolf zur Ziege und dann wieder alleine zurück, hat er oder sie nicht die richtige Lösung gefunden – das Lösungswort gehört nicht zur Sprache der korrekten Lösungen!

Anschaulich kann das gesamte Prüfverfahren mit Hilfe eines Deterministischen Endlichen Automaten (hier in bildhafter Form) dargestellt werden. Die Kreise symbolisieren die jeweils zulässigen Situationen an den beiden Flußufern, die Pfeile symbolisieren die Überfahrten, wobei die Bootsinhalte mit kleinen Buchstaben abgekürzt werden (m - Mensch alleine, w - Mensch mit Wolf, z - mit Ziege, k - mit Kohlkopf). Korrekt sind dann alle Überfahrt-Folgen, die – beginnend beim Start-Zustand MWZK-0 – zum Endzustand 0-MWZK führen, der im nachfolgenden Bild mit einem doppelten Rand gezeichnet ist.



Ein Endlicher Automat mit Mensch, Ziege, Wolf und Kohlkopf

6.3.1.2. Definition (Deterministischer Endlicher Automat)

Ein *Deterministischer Endlicher Automat* (DEA) ist ein mathematisches Modell eines Systems mit endlich vielen Zuständen und diskreten Inputs (Zeichen für Zeichen).

Dabei hat ein Endlicher Automat kein "Gedächtnis", das heißt, er geht alleine aufgrund der Information seines momentanen Zustandes und des nächsten gelesenen Zeichens in den folgenden Zustand über.

Formal: Ein Deterministischer Endlicher Automat ist ein 5-Tupel (Z,A,δ,q_0,E) bestehend aus

1. Z: einer endlichen Menge von Zuständen,
2. A: einem endlichen Alphabet,
3. δ: einer Übergangsfunktion, die einem Zustand $q \in Z$ und einem gelesenen Zeichen $a \in A$ genau einen Zustand q' zuordnet: $q' = \delta(q,a)$.
4. q_0: einem Startzustand (aus Z) sowie
5. E: einer Menge der (möglichen) Endzustände (Teilmenge von Z).

Zu einem Deterministischen Endlichen Automaten assoziieren wir ein Übergangsdiagramm und eine Übergangstafel, die beide darstellen, welche Zuordnungsvorschriften die Übergangsfunktion δ enthält. Beide Darstellungsformen werden anhand des nachfolgenden Beispiels erläutert.

6.3.1.3. Beispiel und Definition (DEA, Übergangsgraph)

Sei ein Deterministischer Endlicher Automat (Z,A,δ,q_0,E) gegeben mit

$Z := \{ q_0, q_1, q_2, q_3 \}$,

$A := \{ a, b \}$,

$E := \{ q_2 \}$,

sowie der Übergangsfunktion δ wie nachstehend in den beiden Darstellungsformen *Übergangstafel* und *Übergangsgraph* angegeben.

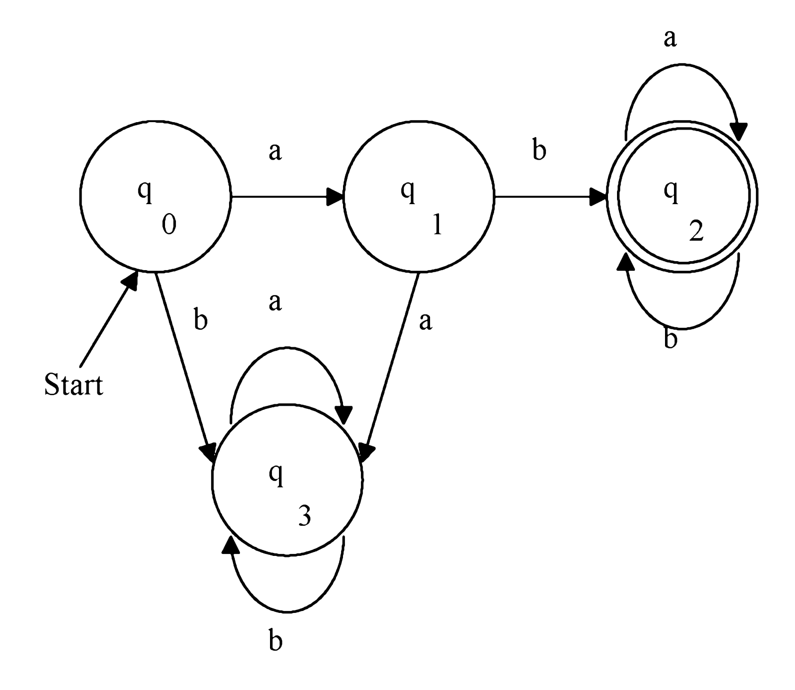
Darstellungsform 1: **Die Übergangstafel**

Bei der Übergangstafel eines Deterministischen Endlichen Automaten werden die Zeichen des Alphabets aufgetragen gegen die Zustände aus Z. In die betreffenden Felder der Matrix wird jeweils eingetragen, in welchen Zustand der DEA wechselt, wenn er sich in einem bestimmten Zustand befindet und ein Zeichen liest. Die möglichen Endzustände werden dabei mit einem Plus-Zeichen "+" versehen.

Z \ A	***a***	***b***
q_0	q_1	q_3
q_1	q_3	q_2
$+q_2$	q_2	q_2
q_3	q_3	q_3

Darstellungsform 2: **Der Übergangsgraph**

Beim Übergangsgraphen eines Deterministischen Endlichen Automaten wird, wie der Name schon sagt, graphisch dargestellt, in welcher Weise der DEA von einem Zustand in einen anderen wechselt. Die folgende Skizze stellt denselben Deterministischen Endlichen Automaten dar wie obige Übergangstafel.



Ein Übergangsgraph (Übergangsdiagramm)

Der hier gezeigte Automat akzeptiert (=erkennt) alle Worte, die mit dem String *ab* beginnen. Die Zustände q_2 und q_3 werden *absorbierend* genannt, da der Automat aus diesen Zuständen unabhängig von der Eingabe nicht mehr herauskommen kann. Einen absorbierenden Nicht-Endzustand wollen wir anschaulich *Müllzustand* nennen.

6.3.1.4. Definition (erkannte Sprache)

Ein Deterministischer Endlicher Automat M *akzeptiert* ein Wort w über dem Alphabet A genau dann, wenn er, ausgehend vom Startzustand q_0, bei zeichenweiser Abarbeitung des Wortes w sich schließlich in einem Endzustand befindet. Andernfalls wird w nicht akzeptiert. Die von M *erkannte Sprache L(M)* ist die Menge aller Worte, die M akzeptiert.

6.3.1.5. Konvention

Gelegentlich werden (aus Gründen der Übersichtlichkeit) Müllzustände nicht mit in das Übergangsdiagramm eines Endlichen Automaten eingezeichnet. Ein solcher (unvollständiger) Automat ist dementsprechend so zu interpretieren, dass jeder nicht eingezeichnete Übergang in einen formal noch zu ergänzenden Müllzustand führt. Das entsprechend eingelesene Wort gehört damit jedenfalls nicht mehr zu der Sprache, die der Automat erkennt.

6.3.2. Nichtdeterministische Endliche Automaten

Neben den Deterministischen Endlichen Automaten gibt es noch eine (kleine) Erweiterung, die sogenannten Nichtdeterministischen Endlichen Automaten, bei denen in Abweichung von den DEA die Übergangsfunktion δ nicht eindeutig zu sein braucht. Mathematisch präziser: die Übergangsfunktion ist im nichtdeterministischen Fall mengenwertig.

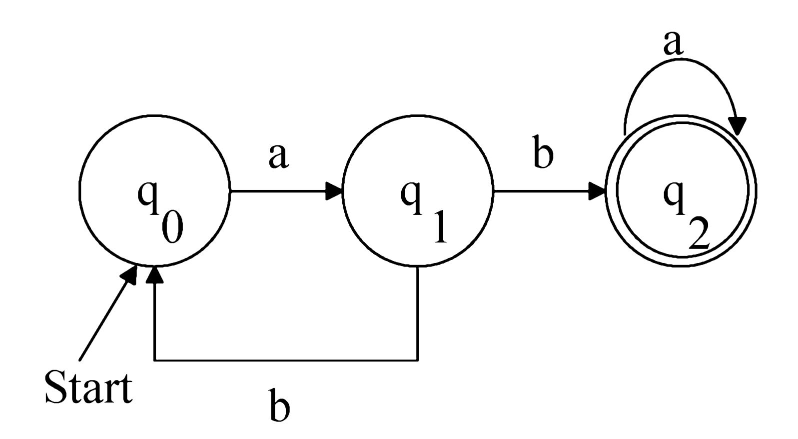
6.3.2.1. Definition (Nichtdeterministischer Endlicher Automat)

Ein *Nichtdeterministischer Endlicher Automat* (*NEA*) ist ein Fünftupel (Z,A,δ,q_0,E) mit

1. Z: einer endlichen Menge von Zuständen,
2. A: einem endlichen Alphabet,
3. δ: einer Übergangsfunktion, die einem Zustand $q \in Z$ und einem gelesenen Zeichen $a \in A$ eine (nichtleere) Menge von Zuständen Q' zuordnet: $Q' = \delta(q,a)$.
4. q_0: einem Startzustand (aus Z)   sowie
5. E: einer Menge der (möglichen) Endzustände (Teilmenge von Z).

Die Definition der von einem Automaten akzeptierten oder erkannten Sprache gilt entsprechend auch für Nichtdeterministische Endliche Automaten.

Der Unterschied zur Definition eines Deterministischen Endlichen Automaten liegt also lediglich in der Festlegung der Übergangsfunktion. Folgende Skizze zeigt das Übergangsdiagramm eines Nichtdeterministischen Endlichen Automaten. (Auch hier wird die oben erwähnte "Müll-Konvention" verwendet.)



Ein Übergangsgraph (Übergangsdiagramm)

Weitere Übungen und Betrachtungen ergeben, dass zum Entwerfen eines Automaten das Konzept des NEA äußerst praktisch ist, gleichzeitig jedoch nicht mehr Sprachen dadurch erkannt werden können. Anders formuliert: die Menge der von Nichtdeterministischen Endlichen Automaten erkennbaren Sprachen ist identisch mit der Menge der von Deterministischen Endlichen Automaten erkannten Sprachen. Dies geht sogar soweit, dass es

einen Überführungsmechanismus gibt, der zu einem NEA einen zugehörigen Deterministischen Endlichen Automaten konstruiert, der also dieselbe Sprache erkennt wie der Ausgangsautomat!

6.3.2.2. Satz (Überführungsalgorithmus NEA→DEA)

Nachstehend wird ein Algorithmus skizziert, der zu einem NEA den (bzw. einen) zugehörigen DEA konstruiert.

Eingabe	Ein Nichtdeterministischer Endlicher Automat N := (Z,A,d,q_0,E)
Ausgabe	Ein Deterministischer Endlicher Automat D := (Z',A,d',Q_0,E'), der äquivalent zu N ist, d. h. L(N)=L(D).
Methode	Jeder Zustand Q von D ist die Menge der Zustände, die N bei derselben Eingabe erreicht haben könnte.
Verfahren	Initialisierung: Q_0 := { q_0 } ist Startzustand von D; Q_0 sei unmarkiert.

SOLANGE es einen unmarkierten Zustand Q von D gibt:
  BEGIN
  FÜR JEDES *a* AUS A:
    BEGIN
    Sei Q' die Menge der Zustände des NEA, zu denen es für *a* einen
      Übergang von einem Zustand q aus Q gibt;
    FALLS Q' noch kein Zustand von D ist, DANN mache Q' zu
      einem unmarkierten Zustand von D;
    Füge einen Übergang von Q nach Q' mit der Beschriftung *a* in den
      Automaten D ein, sofern dieser nicht schon existiert
    END;
  Markiere Q
  END

Anmerkung: Jedes Q (= Zustand von D = Menge von Zuständen von N), das einen Endzustand von N enthält, ist ein Endzustand von D.

6.4. Compilerbau

In diesem Abschnitt ist unser Ziel, die syntaktische Analyse von Programmtexten zu realisieren. Unser Weg wird der Transfer von Syntaxdiagrammen (über die Backus-Naur-Form) hin zu Erkennungsprozeduren (sogenannten Parsern) sein.

6.4.1. Einführung

Sind beispielsweise die Ableitungsregeln S ::= AB, A ::= 1 und B ::= 2 gegeben, so ist es offensichtlich kein besonders aufregender Unterschied, den (einzigen) Satz der erzeugten Sprache 12 einmal über den Ableitungsweg S ⇒ AB ⇒ A2 ⇒ 12 und ein anderes Mal über den Weg S ⇒ AB ⇒ 1B ⇒ 12 herzuleiten. Diese Überlegung führt uns zur nächsten Definition, die solche eher banalen Unterscheidungen für unsere weiteren Untersuchungen ausklammert.

Ist eine Grammatik G gegeben, so heißt eine Ableitung *Linksableitung* (*left-most derivation*), wenn in jeder Satzform das am weitesten links stehende Nichtterminalzeichen abgeleitet wird.

Dieser Begriff der Linksableitung bedeutet für unsere oben erwähnten Beispielregeln, dass der Ableitungsweg S ⇒ AB ⇒ 1B ⇒ 12 die Linksableitung für den Satz 12 ist, denn jede Satzform (im vorliegenden Falle also S, AB, 1B) wird "von links" weiter abgeleitet. (Bei dem hierzu symmetrischen Weg spricht man konsequenterweise dann von einer Rechtsableitung.)

6.4.1.1. Beispiel (Ableitungen aus Regeln)

Es seien die nachfolgenden beiden Grammatiken G_1 und G_2 vorgegeben, die beide eine (kleine) Sprache von Expressions (Ausdrücken) erzeugen. (Aus Gründen der Übersichtlichkeit werden jeweils nur die Produktionsregeln benannt. Die Mengen der Nichtterminal- und der Terminalzeichen sind { S, E, T, F, U } bzw. { +, *, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }. Das Startsymbol heißt, wie gewohnt, wieder S.)

G_1 habe die Produktionsregeln:

S ::= E

E ::= E "+" T | T [115]

T ::= T "*" F | F

F ::= "(" E ")" | U

U ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

(Dabei steht E für expression, T für term, F für factor und U für unit, hier also eine einstellige Dezimalzahl, allgemeiner könnte dies ein integer- oder real-Literal sein.)

---

[115] Regeln der Form E ::= E "+" T heißen linksrekursiv.

G_2 habe die folgenden Produktionsregeln:

S ::= E

E ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
E "+" E | E "*" E | "(" E ")"

Geben wir nun an, wie der (von beiden Grammatiken erzeugte) Satz *1+2*3* abgeleitet werden kann.

1. Zu G_1: S ⇒ E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ U+T ⇒ 1+T ⇒ 1+T*F ⇒ 1+F*F ⇒ 1+U*F ⇒ 1+2*F ⇒ 1+2*U ⇒ 1+2*3

2. Zu G_2: Hier finden wir zwei Ableitungswege:
(1) S => E => E+E => 1+E => 1+E*E => 1+2*E => 1+2*3
und
(2) S => E => E*E => E+E*E => 1+E*E => 1+2*E => 1+2*3

Beachten Sie bitte, dass beide Ableitungswege Linksableitungen sind und trotzdem diese zwei unterschiedlichen Wege existieren!

Weg (1) führt unter Beachtung der üblichen Hierarchiebildung bei arithmetischen Ausdrücken semantisch zu dem Ausdruck 1+(2*3), während Weg (2) in dem Rechenausdruck (1+2)*3 mündet. (Klar?)

6.4.1.2. Definition (Mehrdeutigkeit)

Eine Grammatik G heißt *mehrdeutig* (*ambiguous*), wenn es (mindestens) einen Satz der von G erzeugten Sprache L(G) gibt, der mehr als eine Linksableitung besitzt [bzw. zu dem es keinen eindeutigen Ableitungsbaum gibt].

Dabei entsteht ein Ableitungsbaum zu einem Satz der Typ-2-Sprache L(G), indem – ausgehend von der Wurzel, die durch das Startsymbol S gebildet wird – von Ebene zu Ebene Regelanwendungen dargestellt werden: An jedes Nichtterminalzeichen werden als Kindknoten die Elemente der rechten Seite der in diesem Schritt angewendeten Regel angehängt.

6.4.1.3. Beispiel (Ableitungsbäume)

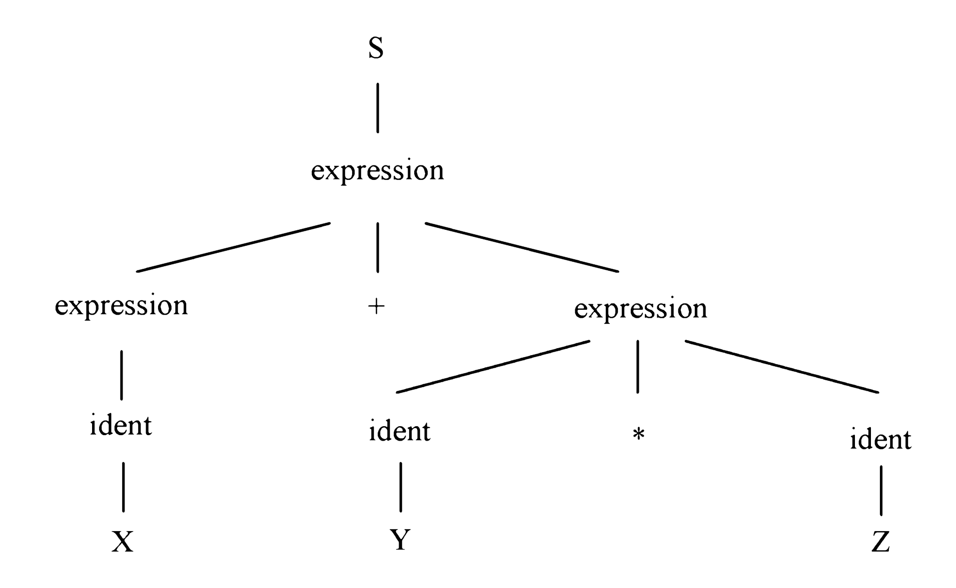
Es seien die folgenden Produktionsregeln (in Backus Naur Form) gegeben.

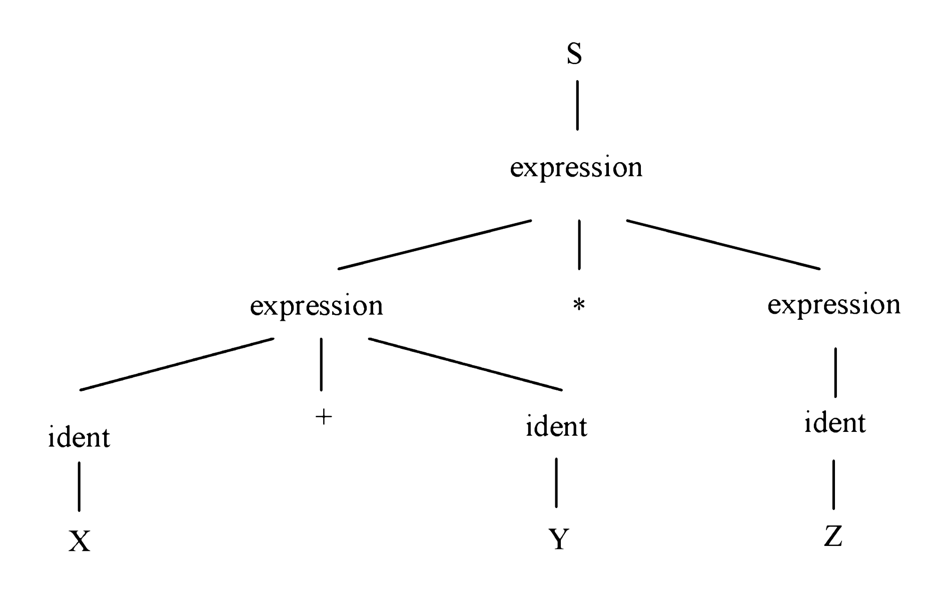
S ::= <expression>

<expression> ::= <expression> "+" <expression> |
<expression> "*" <expression> |
<identifier>

<identifier> ::= "X" | "Y" | "Z"

Dann ist der Satz X+Y*Z gleich auf zwei Weisen abzuleiten, wie die nachstehenden Ableitungsbäume zeigen.





Beispiel für Mehrdeutigkeit

In der Praxis ist Mehrdeutigkeit möglichst zu vermeiden, z. B. durch das Aufstellen von Konventionen, wie wir es von der bekannten Punkt-vor-Strich-Regel kennen. Das heißt: im obigen Beispiel würde ein Pascal-Compiler den erstgenannten Ableitungsbaum erzeugen, da in ihm die Grammatik G_1 umgesetzt ist, bei der die Operatoren gemäß der Vorrangstellung auf unterschiedlichen Ebenen auftreten.

Unser obiges Beispiel G_2 ist also eines für Mehrdeutigkeit, das eine solche Konvention erforderlich macht.

6.4.1.4. Bemerkung: Mehrdeutigkeit ist nicht entscheidbar

Mehrdeutigkeit ist nicht entscheidbar! Das heißt: es gibt keinen allgemein gültigen Algorithmus, der zu einer vorgegebenen kontextfreien Grammatik entscheiden kann, ob sie Mehrdeutigkeit beinhaltet!

Wie bereits gesagt, wird als Top-Down-Methode die Ableitung vom Startsymbol S ausgehend bezeichnet. – Was könnte ein Parser (= eine Erkennungsprozedur) bei Top-Down-Analyse und einer (ziemlich) beliebigen Typ-2-Grammatik versuchen? Sehen wir uns die obige Grammatik G_1 an; nehmen wir wieder den Beispielsatz 1+2*3 her. Es ist also abzuarbeiten das Startsymbol S; im sogenannten Eingabeband (≅ Quelltext unserer Programmiersprache in späteren Beispielen) steht 1+2*3.

Schematisch:

Regel	**Satzform**	**Eingabeband**
	S	1+2*3
S ::= E	E	1+2*3
E ::= E + T [1])	E+T	1+2*3
E ::= E + T	E+T+T	1+2*3
	... *Endlosschleife!* ...	

[1]) Annahme hierbei: der Parser versucht, entsprechend der Nennung in der Grammatik, diese Regel zuerst!

Top-Down-Analyse-Ansatz für 1+2*3

Das heißt: weil die linksrekursiven Regeln[116] (für E und T) in G_1 zuerst genannt werden, läuft unsere erste Parserversion in eine Endlosschleife. (Eine solche Rekursion kann im übrigen auch viel "versteckter" in Form einer sogenannten indirekten Rekursion auftreten!)

Nächster Ansatz ist also die „Reparatur", die darin besteht, alle linksrekursiven Ableitungsregeln nach hinten zu setzen. Die Regeln von G_1 haben dann diese Gestalt:

S ::= E

E ::= T | E "+" T

T ::= F | T "*" F

F ::= "(" E ")" | U

U ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

[116] Eine Ableitungsregel der Form X ::= XY heißt *linksrekursiv*, da das abzuleitende Nichtterminalzeichen X – die linke Seite – auch auf der rechten Seite der Produktionsregel, und zwar als allererstes, wieder auftritt.

Aber auch hier ergeben sich Probleme: spielen wir die Top-Down-Analyse von 1+2*3 nochmals durch mit diesen Regeln.

Regel	**Satzform**	**Eingabeband**
	S	1+2*3
S ::= E	E	1+2*3
E ::= T	T	1+2*3
T ::= F	F	1+2*3
F ::= U	U	1+2*3
U ::= 1	1	1+2*3
ε	_	_+2*3

Hiermit wäre die "1" korrekt erkannt worden, alle Nichtterminalzeichen wurden abgearbeitet, aber: im Eingabeband steht noch der unverarbeitete Rest "+2*3"!

Weiterer Top-Down-Analyse-Ansatz für 1+2*3

Im hier gezeigten Fall wäre also ein Backtracking erforderlich: das Parser-Programm muss, an einer Stelle wie im obigen Bild beschrieben angekommen, Schritt für Schritt zurückgehen („*backtracking*" durchführen), bis es mit alternativen Ableitungsregeln weitermachen kann! Dies bedeutet in der Praxis einen (sehr) großen Aufwand bereits für die Analyse eines solch einfachen Satzes wie *1+2*3* !

Gehen wir davon aus, dass eine Parser-Prozedur (wie in obigen Bildern angedeutet) den zu analysierenden (Quell-)Text streng sequentiell liest, dann können wir uns das soeben aufgezeigte Problem auch anhand des prominenten *if-then-else*-Problems ansehen.

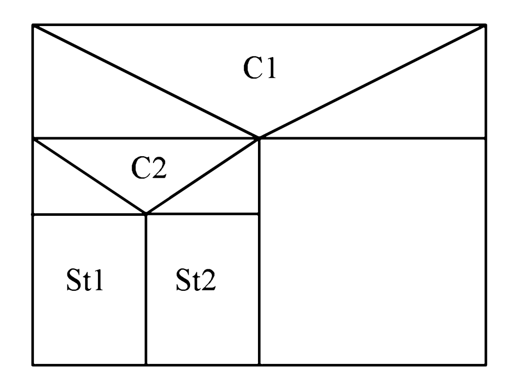
6.4.1.5. Beispiel: Dangling-Else-Problem.

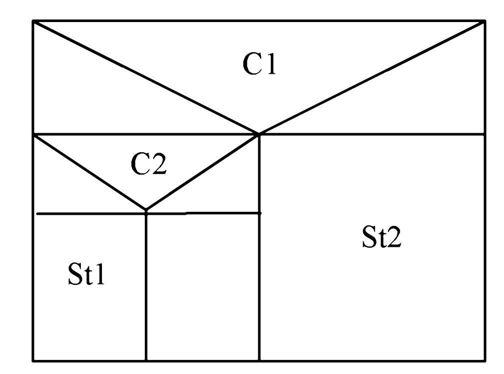
In einer Programmiersprache sei die Syntax für ein if-Statement folgendermaßen (in Backus-Naur-Form) vorgegeben.

```
<stmt> ::=    if <cond> then <stmt>  |
              if <cond> then <stmt> else <stmt> |
              <other-stmt>
```

Sind C1 und C2 boolesche (logische) Ausdrücke (*conditions*) und St1 und St2 zwei konkrete Anweisungen (*statements*), dann ist das if-Statement *if C1 then if C2 then St1 else St2* mehrdeutig, was nachzuvollziehen ist, wenn die Anweisung als Struktogramm dargestellt wird. Entweder ist der erste Ableitungsschritt der zur Sequenz *if C1 then <stmt>* mit der weiteren Ableitung *if C1 then if C2 then St1 else St2*, oder aber die erste Ableitung führt zu *if*

C1 then <stmt> else St2 und der nächste Schritt zu *if C1 then if C2 then St1 else St2*.





Zwei Struktogramme zur Sequenz *if C1 then if C2 then St1 else St2*

Die Lösung dieses Problems, d. h. die Angabe anderer Ableitungsregeln, die dieselben if-Statements generieren, gleichzeitig aber ohne Mehrdeutigkeit auskommen, soll an dieser Stelle nicht verraten werden. Dies ist vielmehr Gegenstand der Übungsaufgabe Fehler 124: Feld endet nicht richtig.

Insgesamt ist also Ziel unserer ganzen Bemühungen, eine "sackgassenfreie" Top-Down-Analyse durchführen zu können. Das Problem im Dangling-Else-Fall ist, dass bei einer sequentiellen ("von oben nach unten") Lektüre eines Quelltextes nicht festgestellt werden kann, zu welchem *if* das eine *else* gehört, da es zwei *if*s gibt.

6.4.2. First- und Follow-Mengen

Nach dem oben Gesagten ist also einleuchtend, dass zu einer Ableitungsregel der Form

$$X ::= \sigma_1 \mid \sigma_2 \mid ... \mid \sigma_n \quad \text{mit } X \in N \text{ und } \sigma_i \in (N \cup T)^*$$

klar sein muss, welches (eindeutige) σ_i für die nächste konkrete Ableitung genommen werden kann.

6.4.2.1. Beispiel (First- und Follow-Problem)

Sei G := (N,T,S,P) eine Grammatik mit N := { S, A, B }, T := { a, b, c } und den Produktionsregeln P := { S ::= A | B, A ::= cA | a, B ::= cB | b }. – Dann ist die von G erzeugte Sprache L(G) die Menge aller Sätze der Form $c^n a$ bzw. $c^n b$ (mit $n \geq 0$).
Aber: bei der Analyse des Satzes ccca ist erst beim Lesen des Zeichens a klar, dass bei der allerersten Ableitung des Startsymbols S die Regel S ::= A verwendet werden musste!

Dieses Beispiel legt den Wunsch nahe, dass bei Ableitungen der Form

$$X ::= \sigma_1 \mid \sigma_2 \mid ... \mid \sigma_n \quad \text{mit } X \in N \text{ und } \sigma_i \in (N \cup T)^*$$

der Anfang jeder rechten Seite σ_i zu einem eindeutigen Terminalzeichen führt, so dass durch Lesen nur eines weiteren Terminalzeichens die Parser-Prozedur weiß, welche Ableitungsregel im nächsten Schritt anzuwenden ist.

Formal: Lassen sich z. B. σ_1 (in mehreren Schritten) zur Terminalzeichenfolge $t_1t_2...t_n$ und σ_2 (evtl. ebenfalls in mehreren Schritten) zur Terminalzeichensequenz $u_1u_2...u_n$ ableiten, dann muss $t_1 \neq u_1$ gelten. – Diese Forderung führt zur folgenden Definition.

6.4.2.2. Definition (First-Menge)

Zu $\sigma \in (N \cup T)^*$ wird die *First-Menge first(σ)* definiert wie folgt.

Für den Spezialfall $\sigma = \varepsilon$ wird $\text{first}(\sigma) := \emptyset$ definiert;
ist $\sigma \neq \varepsilon$, so wird definiert

$$\text{first}(\sigma) := \{ t \in T \mid \sigma \overset{*}{\Rightarrow} t\sigma', \sigma' \in (N \cup T)^* \}.$$

Hierbei steht $\sigma^* \Rightarrow t\sigma'$ für eine Ableitung in null oder endlich vielen Schritten. Das heißt z. B. $\text{first}(t) = \{ t \}$ für $t \in T$.

6.4.2.3. Beispiel (First-Mengen)

Sei $G := (N,T,S,P)$ eine Grammatik mit $N := \{ S, A \}$, $T := \{ t, + \}$, dem Startsymbol S und den Produktionsregeln $P := \{ S ::= tA, A ::= A + t, A ::= \varepsilon \}$. Dann ist die von G erzeugte Sprache $L(G) = \{ t(+t)^n \mid n>0 \}$. – Die auftretenden first-Mengen sind

$$\text{first}(S) = \{ t \}, \text{first}(A) = \text{first}(A + t) = \{ + \} \text{ und } \text{first}(\varepsilon) = \emptyset.$$

6.4.2.4. Anmerkung (Typ-2-Grammatiken und First-Mengen)

Sie werden bemerken, dass wir einerseits Typ-2-Grammatiken (kontextfreie Grammatiken) betrachten wollten, gleichzeitig aber im obigen Beispiel 6.4.2.3. eine ε-Ableitung auftritt. Solche ε-Ableitungen sollen im Folgenden die einzige Ausnahme von der strengen Typ-2-Vorgabe sein.

Ist ε nicht selbst Element der Sprache, dann lassen sich Regeln der Form $X ::= \alpha Y \beta$, $Y ::= \gamma \mid \varepsilon$ leicht umformen zu $X ::= \alpha Y \beta \mid \alpha\beta$, $Y ::= \gamma$. Wie wir auch bei unserem später zu betrachtenden Parser sehen werden, verzichtet man in der Praxis auf diesen Mehraufwand.

6.4.2.5. Beispiel (First-Mengen)

Nehmen wir die Grammatik G aus Beispiel 6.4.2.1. Dann ist $\text{first}(A) = \{ a, c \}$ und $\text{first}(B) = \{ b, c \}$. Das dort dargestellte Entscheidungsproblem resultiert daraus, dass beide Nichtterminalzeichen identische Terminalzeichen in ihren First-Mengen besitzen.

6.4.2.6. Beispiel (Mehrdeutigkeit der Ableitungen)

Sei $G := (N,T,S,P)$ mit $N := \{ S, A, V \}$, $T := \{ +, 0, 1 \}$ und den Produktionen

$$P := \{ S ::= 1VA, V ::= + \mid \varepsilon, A ::= +0 \mid +1 \mid 0 \mid 1 \}.$$

Im hier vorliegenden Fall tritt ein neues Problem auf: der Satz 1+1 ist (wieder) mehrdeutig (links-)ableitbar:

$S \Rightarrow 1VA \Rightarrow 1{+}A \Rightarrow 1{+}1$ bzw. $S \Rightarrow 1VA \Rightarrow 1{\in}A \Rightarrow 1{\in}{+}1$,

und das ist ebenfalls 1+1! Diesmal resultiert das Problem aus der ε-Ableitung für V!

Beispiel 6.4.2.6. führt uns zu einer weiteren Definition, die speziell im Falle von ε-Ableitungen zum Zuge kommen wird.

6.4.2.7. Definition (Follow-Menge)

Zu einem Nichtterminalzeichen X wird die *Follow-Menge follow(X)* definiert als die Menge all der Terminalzeichen, die in Satzformen rechts von X auftreten können, formal:

$$\text{follow}(X) := \{ t \in T \mid S \overset{+}{\Rightarrow} \sigma_1 X t \sigma_2,\ \sigma_1,\sigma_2 \in (N \cup T)^* \}.$$

In Beispiel 6.4.2.6. ist das erwähnte Problem also formal dadurch beschreibbar, dass das Nichtterminalzeichen V, für das es eine ε-Ableitung gibt, sowohl in seiner First-Menge first(V), als auch in seiner Follow-Menge follow(V) das Terminalzeichen + beinhaltet. Deswegen kann bei sequentieller Analyse nicht entschieden werden, ob V ::= ε oder V ::= + konkret verwendet werden muss.

6.4.3. LL(1)-Grammatiken

Die bisherigen Betrachtungen führen uns zur Formulierung zweier Regeln.

Sei G := (N,T,S,P) eine (Fast-)Typ-2-Grammatik, d. h. zusätzlich zu den Typ-2- Regeln seien ε-Ableitungen erlaubt.

6.4.3.1. Regel 1

Existieren für ein Nichtterminalzeichen X alternative Ableitungsregeln,

$$X ::= \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n \quad \text{mit } X \in N \text{ und } \sigma_i \in (N \cup T)^*,$$

so sind die First-Mengen der rechten Seiten dieser Ableitungsregeln paarweise disjunkt. Formal: für alle $i \neq k$ ist $\text{first}(\sigma_i) \cap \text{first}(\sigma_k) = \emptyset$.

6.4.3.2. Regel 2

Tritt (in Erweiterung des Begriffes kontext-frei) eine ε-Ableitung des Nichtterminalzeichens X, d. h. X ::= ε, auf, so sind die First- und die Follow-Menge von X disjunkt. Formal: Ist X ::= ε in P enthalten, so ist $\text{first}(X) \cap \text{follow}(X) = \emptyset$.

Wenn diese beiden Regeln für eine Grammatik G erfüllt sind, dann können wir unser Vorhaben, eine Parser-Prozedur für die Sprache L(G) zu implementieren, weiterverfolgen.

6.4.3.3. Definition (LL(1)-Grammatik)

Eine Grammatik G heißt *LL(1)*-Grammatik ["*left-to-right-scanning, leftmost derivation, one symbol lookahead*"], wenn ihre Ableitungsvorschriften P den oben genannten Regeln 1 und 2 genügen.

Sprachen, die von solch einer Grammatik erzeugt werden können, bekommen ebenfalls einen eigenen Namen.

6.4.3.4. Definition (LL(1)-Sprache)

Eine Sprache L heißt *LL(1)-Sprache*, wenn es eine LL(1)-Grammatik G gibt, die L erzeugt, d. h. für die L = L(G) gilt.

6.4.3.5. Beispiel (LL(1)-Grammatik)

Sei G := (N,T,S,P) gegeben mit N := { S, A, B }, T := { 0, 1 } und den Produktionsregeln P := { S ::= 1A1 | 0B1, A ::= 0A | 1, B ::= 0 | ε }.

Dann sind die LL(1)-Bedingungen zu prüfen:

Die Mengen first(1A1) = { 1 } und first(0B1) = { 0 } sind disjunkt, damit sind die beiden Regeln für S erfüllt; die First-Mengen first(0A) = { 0 } und first(1) = { 1 } sind ebenfalls disjunkt.

Für das Nichtterminalzeichen B muss (wegen der ε-Ableitung) die Regel 2 überprüft werden:

first(B) ist offenbar (?) die Menge { 0 }, follow(B) lässt sich nur durch Betrachtung der gesamten Grammatikregeln erfassen: die Produktionsregel S ::= 0B1 ist die einzige, in der das Nichtterminalzeichen B auf einer rechten Seite auftritt. Daher ist follow(B) abzulesen als { 1 }. Somit ist first(B) ∩ follow(B) = ∅.

Damit ist G eine LL(1)-Grammatik. Infolgedessen ist natürlich L(G) eine LL(1)-Sprache.

6.4.4. Aufbau eines Parsers für eine LL(1)-Grammatik

Bevor wir uns der etwas umfangreicheren Modellsprache PL/0 (gemäß [Wirth2]) zuwenden, wollen wir elementare Überlegungen anstellen, wie ein Parser, eine Analyseprozedur, arbeiten könnte. Wir bearbeiten die Problematik hier konkret mit den Programmiersprachen Pascal und C. Aus Gründen des hierarchischen Prozedurkonzepts von Pascal besprechen wir im Folgenden diese Version; die C-Variante des Parsers wird anschließend gezeigt.

Sinnvoll ist zunächst eine Prozedur, die wir *GetSym* nennen wollen, die aus dem Eingabestrom (Quelltext) das jeweils nächste Symbol in einer Variablen namens *sym* bereitstellt. Sie realisiert die lexikalische Analysephase.

Aus der Quelltext-Sequenz *if a >= b then b := a* entsteht (beispielsweise in der Notation des in Abschnitt 4.5. ausführlich vorgestellten Parserprogramms) die Symbolfolge *ifsym ident geq ident thensym ident becomes ident*. Dabei sind *ifsym* und *thensym* Namen für die Symbole, die in Pascal durch die Schlüsselwörter *if* und *then* ausgedrückt werden. *ident* steht für Identifier (Bezeichner), *geq* und *becomes* sind Bezeichnungen für die Symbole >= (*greater or equal*) bzw. := (Zuweisungsoperator).

Darüber hinaus brauchen wir eine Symboltabelle, die die zu einem Symbol zusätzlich notwendigen Informationen speichert, z. B. zum Symbol *ident* für einen Bezeichner dessen Namen und Datentyp oder auch ggf. dessen aktuellen Wert.

Desweiteren soll eine (zunächst spartanische) Fehlerbehandlungsprozedur *Error* bereitgestellt werden. Für unsere Zwecke soll eine kurze Fehlermeldung auf dem Bildschirm genügen und die Arbeit des Parsers sofort nach dem ersten aufgetretenen Fehler beendet werden.

Auf die Konstruktion des abstrakten Baumes für weitere Phasen wird hier verzichtet und die im Ansatz in der Variablen *table* (vgl. den Quelltext in Abschnitt 4.5.1) skizzierte Symboltabelle wird nicht aufbewahrt.

Der grundsätzliche Aufbau des Parsers kann dann einigermaßen direkt von der Darstellung der Grammatik in Backus-Naur-Form oder als Syntaxdiagramm abgeschrieben werden (vgl. auch Abschnitt 2.1.10.).

Aus dem Syntaxdiagramm für eine Alternative ("*A oder B*") entsteht eine geschachtelte if-Struktur (oder eine entsprechende case-/switch-Struktur):

```
if sym in first(A) then
  ProcedureA
else if sym in first(B) then
  ProcedureB
else
  Error
```



Dabei sind ProcedureA und ProcedureB natürlich die entsprechenden Erkennungsprozeduren für die Nichtterminalzeichen A und B. (Die Verallgemeinerung auf eine mehrfache Alternative ist naheliegend.)

Das Syntaxdiagramm für eine Aufeinanderfolge (Konkatenation) wird einfach umgesetzt in die Form

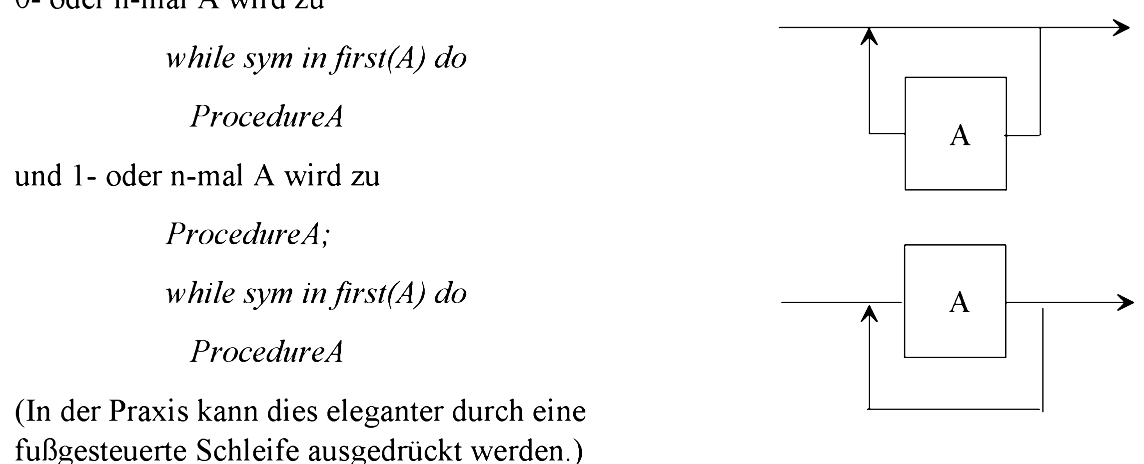


ProcedureA;

ProcedureB;

Dabei ist folgendes zu beachten: da eine eventuelle Fehlererkennung, und damit verbunden eine Fehlermeldung, nicht von der aufrufenden Prozedur veranlaßt wird, sondern in die Prozeduren A respektive B verlagert wird, geht zwangsläufig die Information verloren, in welchem Kontext A bzw. B aufgerufen wurde, also Information, die für die praktische Fehlersuche hilfreich sein kann.

Die Wiederholungsstrukturen (Iterationen) können entsprechend umgesetzt werden:
0- oder n-mal A wird zu



while sym in first(A) do

ProcedureA

und 1- oder n-mal A wird zu

ProcedureA;

while sym in first(A) do

ProcedureA

(In der Praxis kann dies eleganter durch eine fußgesteuerte Schleife ausgedrückt werden.)

Der Spezialfall der Alternative, die Option ("*nichts oder A*"), kann umgesetzt werden in die Form



if sym in first(A) then

ProcedureA

Zu beachten ist hier lediglich, dass keine else-Error-Behandlung folgen kann, da das Syntaxelement A nicht vorkommen <u>muss</u>!

Schließlich wird ein "rundes" Syntaxdiagrammsymbol, das "für sich selbst" steht, entsprechend umgesetzt:

if sym=x then

GetSym { nächstes Symbol holen }

else

Error

6.4.4.1. Beispiel (XParser)

Sehen wir uns die obigen Erläuterungen konkret umgesetzt anhand einer kleinen Sprache ("*Ausdrücke mit X*") an. In Backus-Naur-Form sei die erzeugende Grammatik G := (N,T,S,P) vorgegeben wie folgt: N := { S, A }, T := { x , + , (,) }, die Produktionsregeln P seien S ::= A, A ::= x | A { +A }.

```
{ ----------------------------------------------------------------
  XPARSER     Kleiner Parser für Klammerausdrücke mit "x" und "+".
  ----------------------------------------------------------------
}
program XParser(input,output);

const   Alphabet = [ 'x', '(', ')', '+' ];
        firstA   = [ 'x', '(' ];

type    symbol   = char;
var     sym      : symbol;  { letztes gelesenes Zeichen; hier ist
                              speziell Symbol=Zeichen
                            }
{ ---------------------------------------------- ERROR -------- }
procedure Error;
begin
writeln('Fehler!')
end; { Error }
```

```
{ --------------------------------------------- GETSYM ------- }
procedure GetSym;
begin
if eoln(input) then
  begin
  readln;
  writeln
  end; { if eoln }
read(sym);
write(sym)                   { Listing produzieren }
end; { GetSym }
{ --------------------------------------------- S ----------- }
procedure S;
  { ------------------------------------------- A ----------- }
  procedure A;
  begin
  if sym='x' then
    GetSym
  else if sym='(' then
    begin
    GetSym;
    A;
    while sym='+' do
      begin
      GetSym;
      A
      end; { while sym='+' }
    if sym=')' then
      GetSym
    else
      Error                  { keine schließende Klammer gefunden }
    end { sym='(' }
  else
```

```
    Error                   { weder 'x' noch öffnende Klammer gefunden }
  end; { A }
begin { S }
A;
if sym in Alphabet then     { Jedes andere Zeichen wird als  }
  Error;                    { Abschluss akzeptiert.          }
end; { S }

{ ---------------------------------------------- MAIN --------- }
begin
writeln;
writeln('XParser: Eine Erkennungsprozedur für S::=A, A::=x|(A{+A})');
write('> ');
GetSym;
S
end. { XParser }
```

Und hier die entsprechende XParser-Variante in der Programmiersprache (ANSI-)C, die an einzelnen Stellen vom Vorgehen der Pascal-Version abweicht, indem sie insbesondere einen deutlichen Fehlertext ausgibt. (Vgl. auch die Kommentierungen im folgenden Quelltext).

```
/*-----------------------------------------------------------------
  XPARSER     Kleiner Parser für Klammerausdrücke mit "x" und "+".
  ---------------------------------------------------------------
 */

#include <stdio.h>

#define TRUE  (1==1)
#define FALSE (0==1)
```

```
typedef int  boolean;
typedef char symbol;
enum ERRORS { ERR_NO_ERROR,       /* kein Fehler aufgetreten   */
              ERR_NO_RIGHT_PAREN, /* keine schließende Klammer */
              ERR_NO_X_OR_L_PAREN,/* weder 'x' noch ' ('       */
              ERR_INVALID_START,  /* ungültiges Startsymbol    */
              ERR_INVALID_CHAR,   /* unzulässiges Zeichen      */
};

/* Die C-typische Array-Initialisierung wird hier eingesetzt */
/* zur ausführlicheren Ausgestaltung der Error-Routine.      */
char    *ErrorMsg[] =
        { "[ok]",
          "[Keine schließende Klammer gefunden.]",
          "[Weder 'x' noch öffnende Klammer gefunden.]",
          "[Fehlerhaftes Startsymbol gefunden.]",
          "[Inkorrektes Symbol als Abschluss gefunden.]"
        };

symbol  Alphabet[] = { 'x', '(', ')', '+', NULL };
symbol  FirstA[]   = { 'x', '(', NULL };   /* First-Menge von A */
symbol  Sym;                  /* letztes gelesenes Zeichen/Symbol */
int     ErrorCode = 0,
        SymRead = 0;

/*-------------------------------------------- Prototypen ---*/
void Error(int);
boolean IsIn(symbol, symbol *);
void GetSym(void);
void A(void);
void S(void);
```

```
/*--------------------------------------------- main ---------*/
void main(void)
{
   printf(
    "\nxparser: Eine Erkennungsprozedur für die Grammatik S::=A,"
    "\n~~~~~~~~ A::=x|(A{+A}) ; Eingabe mit [Return] beenden.\n");
   printf("> ");
   GetSym();
   S();
   if (ErrorCode!=0) {
      int i;        /* Markierung der mutmaßlichen Fehlerstelle */
      for (i=0; i<=SymRead; i++)
         putchar(' ');
      putchar('^');
   }
   printf("\n%d Symbol(e) gelesen; ErrorCode: %d %s\n",
      SymRead,ErrorCode,ErrorMsg[ErrorCode]);
} /* end main */

/*--------------------------------------------- Error --------*/
void Error(int errcode)
{
   if (ErrorCode==0)
      ErrorCode=errcode;
   /* 1.Fehler protokollieren! */
} /* end Error */

/*--------------------------------------------- IsIn ---------*/
```

```
boolean IsIn(symbol s, symbol *allowed)
{
   while (*allowed)
      if (s == *allowed++)
         return(TRUE);
   return(FALSE);
} /* end IsIn */

/*-------------------------------------------------- GetSym -------*/
void GetSym(void)
{
   Sym=getchar();
   if (ErrorCode==0 && Sym!='\n')
      SymRead++;
} /* end GetSym */

/*-------------------------------------------------- A -----------*/
void A(void)
{
   if (Sym=='x')
      GetSym();
   else if (Sym=='(')
   {
      GetSym();
      A();
      while (Sym=='+')
      {
         GetSym();
         A();
      }
      if (Sym==')')
         GetSym();
      else
```

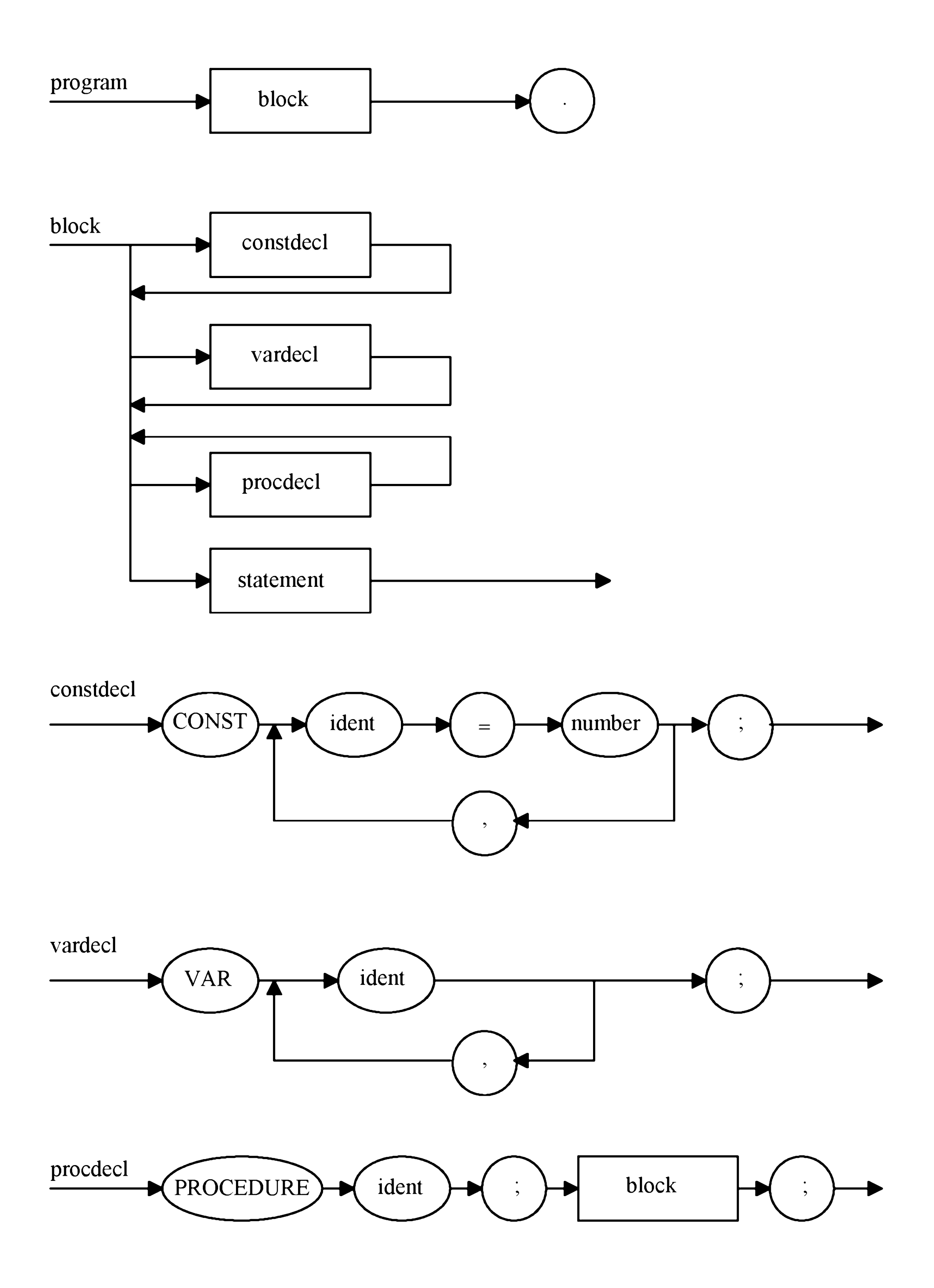
```
            Error(ERR_NO_RIGHT_PAREN);
    } else
        Error(ERR_NO_X_OR_L_PAREN);
} /* end A */

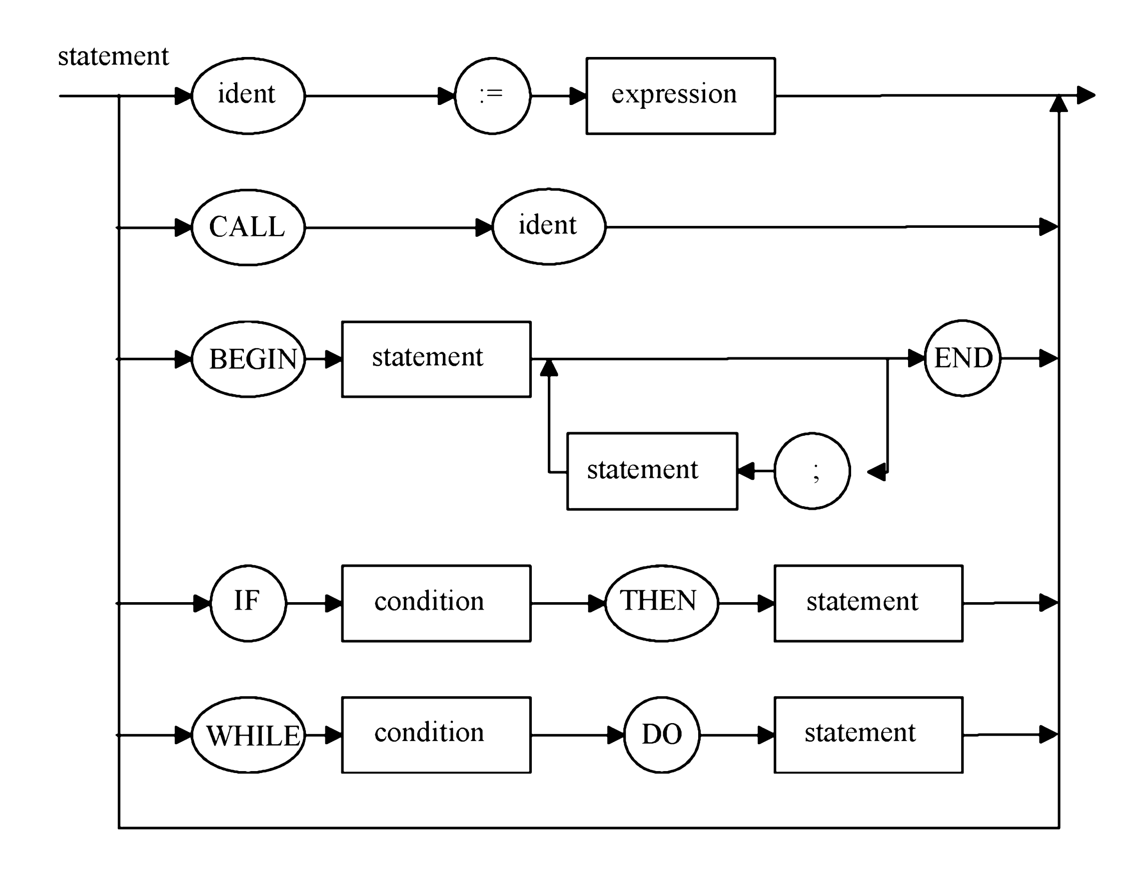
/*---------------------------------------------- S ------------*/
void S(void)
{
    if (IsIn(Sym,FirstA))
        A();
    else
        Error(ERR_INVALID_START);
    if (IsIn(Sym,Alphabet))
        Error(ERR_INVALID_CHAR);
} /* end S */

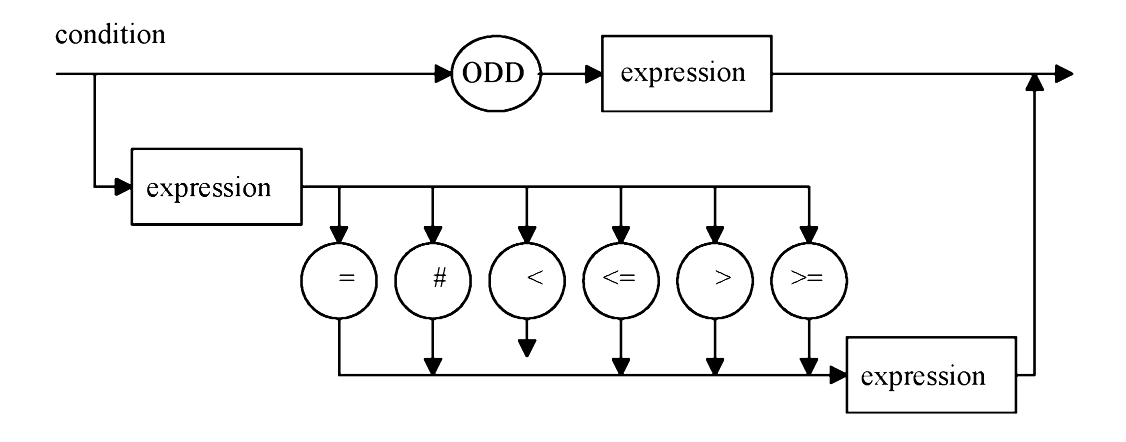
/*-end of file xparser.c----------------------------------------*/
```

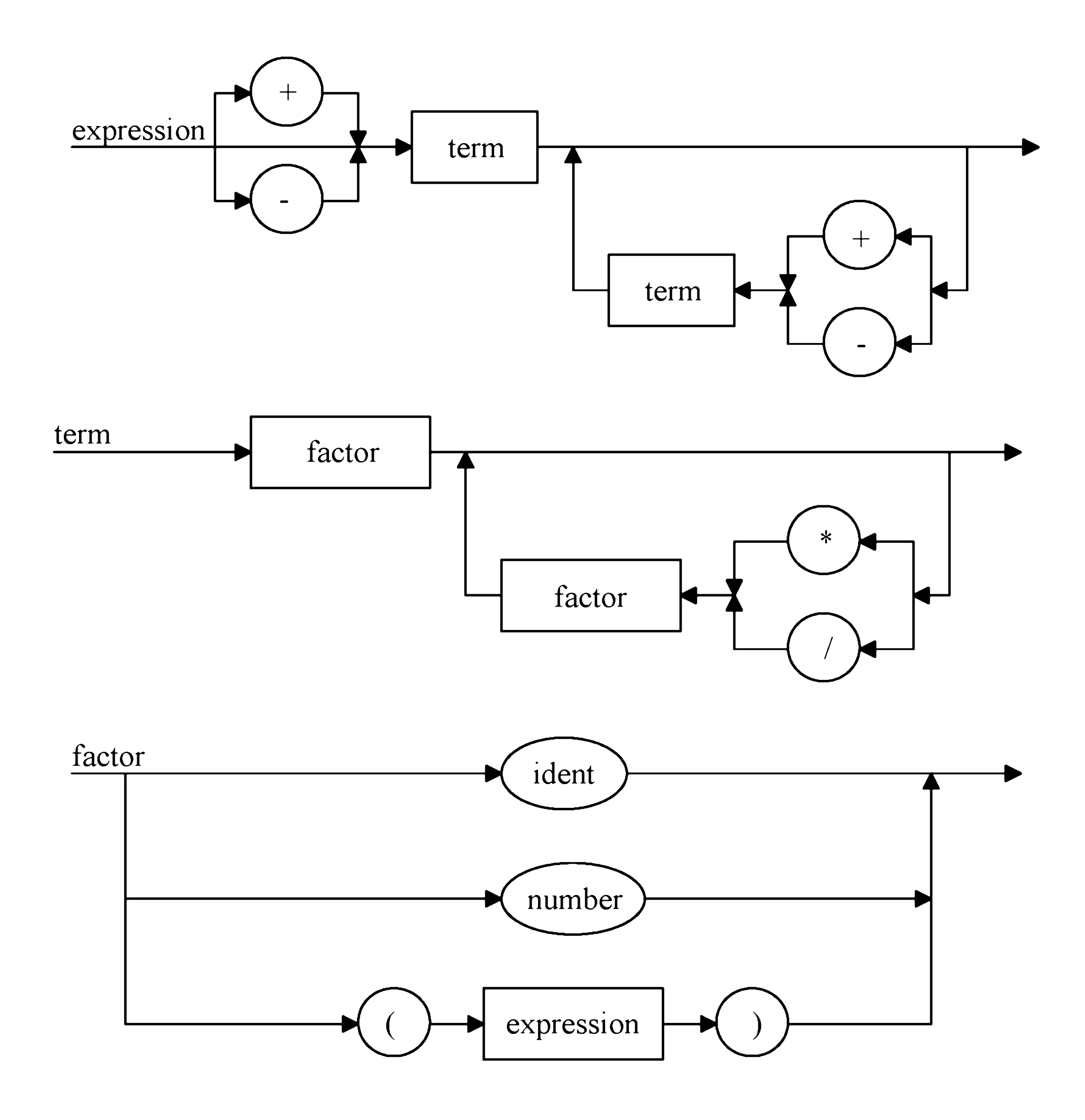
6.4.5. Ein Parser für die Modellsprache PL/0

In diesem Abschnitt wollen wir die vorgestellten Prinzipien anwenden auf einen Parser für die von Wirth eingeführte Modellsprache *PL/0*. Nachfolgend sind die (auf [Wirth2] basierenden) Syntaxdiagramme für PL/0 abgebildet.




statement
ident
:=
expression
CALL
ident
BEGIN
statement
END
statement
;
IF
condition
THEN
statement
WHILE
condition
DO
statement


condition
ODD
expression
expression
=
#
<
<=
>
>=
expression



Die hier vorgestellte Sprache PL/0 kann im Prinzip als kleine Untermenge von Pascal verstanden werden. Sie enthält das Prozedur-Konzept mit seinen geschachtelten Deklarationen und einer leicht variierten Aufrufsyntax (Schlüsselwort `call`) sowie der Lokalität von Variablen. Als Datentyp ist der Einfachheit halber nur der (nicht namentlich benannte) Ganzzahltyp integer realisiert. In der hier vorliegenden Fassung enthält PL/0 keine Ein- und Ausgaberoutinen, da Prozeduren in PL/0 keine Parameterlisten besitzen.

Neben dem Pascal-Quellcode für den PL/0-Parser werden im weiteren auch noch das Blockdiagramm des Programms sowie eine Parservariante in ANSI-C vorgestellt.

6.4.5.1. Quelltext des PL/0-Parsers (Pascal-Version)

Die Syntaxdiagramme für PL/0 lassen sich direkt umsetzen in ein Compiler- bzw. Parser-Programm. Bei diesem handelt es sich um eine Modifikation des von Wirth vorgeschlagenen Programms aus seinem Buch *Compilerbau* [Wirth2].

```
program PL0Parser(input,output);
{*****************************************************************}
{**                                                             **}
{**    PL0.PAS - Standard-Pascal-Version.                       **}
{**    Parser für PL/0.                                         **}
{**                                                             **}
{*****************************************************************}

{ =============================================================== }
{ Deklarationen für den PL/0 Parser                               }
{ =============================================================== }

const
   NORW      =  11;           { "number of reserved words"        }
   TXMAX     = 100;           { Größe der Bezeichner-Tabelle      }
   NMAX      =  14;           { maximale Anzahl von Ziffern in    }
                              {          integer-Zahlen           }
   AL        =  10;           { Akzeptierte Länge von Bezeichnern }
   CHSETSIZE = 256;           { ASCII Zeichensatz                 }
   LINELEN   =  80;
   LETTERS   = ['a'..'z','A'..'Z'];
   ALPHANUMS = ['a'..'z','A'..'Z','0'..'9'];
   NUMBERS   = ['0'..'9'];

type
   symbol    = ( nul, ident, number, plus, minus, times, slash,
                 oddsym, eql, neq, lss, leq, gtr, geq, lparen,
                 rparen, comma, semicolon, period, becomes,
```

```
                 beginsym, endsym, ifsym, thensym, whilesym,
                 dosym, callsym, constsym, varsym, procsym ) ;
   alfa       = packed array [1..AL] of char;
   objekt     = ( constant, variable, proc );

var
   ch         : char;    { jeweils letzter gelesener character       }
   sym        : symbol;  { letztes gelesenes Symbol                  }
   id         : alfa;    { letzter gelesener Bezeichner (identifier) }
   num,                  { letzte  gelesene  Zahl                    }
   cc,                   { "character count"                         }
   ll,                   { "line length"                             }
   zz         : integer;{ Erweiterung: Zeilennummer (fortlaufend)    }
   line       : packed array[1..LINELEN] of char;
   a          : alfa;
   word       : array [1..NORW] of alfa;
   wsym       : array [1..NORW] of symbol;
   ssym       : array [char] of symbol;
   table      : array [0..TXMAX] of
                  record
                     name: alfa;
                     kind: objekt
                  end;
```

```
{ ================================================================}
{ Prozeduren Error und GetSym für den PL/0 Parser               }
{ ================================================================}

procedure Error(n: integer);           { - erste Fehlerprozedur -   }
begin
writeln(' ':cc,'^',n:2);               { Fehlermarkierung im Quell- }
                                       { text mit character count   }
writeln('Fehler gefunden in Zeile ',zz:1,'.');
case n of { Einige Fehlernummern werden hier exemplarisch aufgeführt }
   2 :  writeln('Zahl erwartet!');
   3 :  writeln('Gleichheitszeichen erwartet!');
   4 :  writeln('Bezeichner erwartet!');
   9 :  writeln('Punkt erwartet!');
  11 :  writeln('Nicht deklarierter Bezeichner: ',id);
  31:   writeln('Zahlbereichsüberschreitung! (MAXINT=',MAXINT:1,')');
        { ... hier darf noch beliebig ergänzt werden ... }
  99 :  writeln('Programm unvollständig!')
  else  writeln
end; {case}
halt(1)
end; {Error}
```

```
{-----------------------------------------------------  GetSym  -----}
procedure GetSym;
var i,j,k,neuwert : integer;
    gefunden: boolean;

    procedure GetCh;
    var i: integer;
    begin
    if cc = ll then
       begin
       if eof(input) then
          Error(99);
       zz := zz + 1;
       ll := 0;
       cc := 0;
       write('  ');
       for i:=1 to LINELEN do
          line[i]:=' ';
       while not eoln(input) and (ll < LINELEN-1) do
          begin
          ll := ll+1;
          read(ch);
          write(ch);
          line[ll] := ch
          end;
       writeln;
       ll := ll + 1;
       line[ll]:=' ';
       readln  { read(line[ll]) }
       end;
    cc := cc + 1;
    ch := line[cc];
    end; {GetCh}
```

```
begin {GetSym}
while ch=' ' do
   GetCh;                                 { Leerzeichen werden überlesen }
if ch in LETTERS then
   begin                        {---identifier oder reserviertes Wort---}
   k := 0;
   for i:=1 to 10 do
      id[i] := ' ';
   repeat
      if k < AL then
         begin
         k := k+1;
         id[k] := ch
         end;
      GetCh
   until not (ch in ALPHANUMS);       {--Ende des Identifiers/Wortes--}
   i:=1;
   repeat                             { Suchen / Prüfen:                }
      gefunden := id=word[i];         { Ist es ein reserviertes Wort? }
      if not gefunden then
         i:=i+1
   until gefunden or (i > NORW);
   if gefunden then
      sym := wsym[i]
   else
      sym := ident
   end {ch in LETTERS}          {---identifier oder reserviertes Wort--}
else if ch in NUMBERS then
   begin                                              {---Zahlen---}
   k := 0;
   num := 0;
   sym := number;
```

```
   repeat
      neuwert := ord(ch)-ord('0');
      if num <= (MAXINT - neuwert) div 10 then
         num := 10*num + neuwert
      else
         Error(31);
      k := k+1;
      GetCh
   until not (ch in ['0'..'9']) or (k > NMAX);
   if k > NMAX then
      Error(30)
   end {ch in NUMBERS}                                {---Zahlen---}
else if ch=':' then                  { ab hier: Sonderzeichen       }
   begin
   GetCh;
   if ch='=' then
      begin
      sym := becomes;
      GetCh
      end
   else
      sym := nul
   end {ch=':'}
else if ch='<' then
   begin
   GetCh;
   if ch='=' then
      begin
      sym := leq;
      GetCh
      end
   else
      sym := lss
```

```
   end {ch='<'}
else if ch='>' then
   begin
   GetCh;
   if ch='=' then
      begin
      sym := geq;
      GetCh
      end
   else
      sym := gtr
   end {ch='>'}
else
   begin
   sym := ssym[ch]; { Hiermit wird einem falschen Zeichen das Symbol }
                    { NUL zugewiesen, der eigentlich lexikalische    }
                    { Fehler somit in die syntaktische Phase         }
                    { verlagert.                                     }
   GetCh
   end
end; {GetSym}
{----------------------------------------------------  GetSym  -----}




{ ================================================================= }
{ Prozedur Block (mit Unterprozeduren) für PL/0 Parser              }
{ ================================================================= }

procedure Block(tx: integer);   {---------------------  Block  -----}

   procedure Enter(k: objekt);          { Tabelleneintrag: var,     }
   begin                                { const und proc            }
```

```
tx := tx + 1;
with table[tx] do
   begin
   name := id;
   kind := k
   end {with}
end; {Enter}

function Position(id: alfa): integer;   { Tabelleneintrag finden }
var i: integer;
begin
table[0].name := id;
i := tx;
while table[i].name <> id do
  i := i-1;
Position := i   { entweder Positionsnummer oder 0=nicht gefunden }
end; {Position}

procedure ConstDecl;
begin
if sym=constsym then
   begin
   repeat
      GetSym;
      if sym=ident then
         begin
         GetSym;
         if sym=eql then
            begin
            GetSym;
            if sym=number then
               begin
               Enter(constant);
```

```
            GetSym
            end
         else
            Error(2)
         end
      else
         Error(3)
      end
   else
      Error(4)
until sym<>comma;
if sym=semicolon then
   GetSym
else
   Error(5)
end
end; {ConstDecl}

procedure VarDecl;
begin
if sym=varsym then
   begin
   repeat
      GetSym;
      if sym=ident then
         begin
         Enter(variable);
         GetSym
         end
      else
         Error(4)
   until sym<>comma;
   if sym=semicolon then
```

```
      GetSym
   else
      Error(5)
   end
end; {VarDecl}

procedure ProcDecl(var tx: integer);
begin
if sym=ident then
   begin
   Enter(proc);
   GetSym
   end
else
   Error(4);
if sym=semicolon then
   GetSym
  else
   Error(5);
Block(tx);
if sym=semicolon then
   GetSym
else
   Error(5)
end; {ProcDecl}

procedure Statement;       { Erkennungsprozedur für Anweisung    }
var i: integer;

    procedure Expression;  {                    für Ausdruck     }

        procedure Term;    {                    für Term         }
```

```
    procedure Factor;                  {    für Faktor       }
    var i: integer;
    begin
    if sym=ident then
        begin
        i := Position(id);
        if i=0 then
           Error(11)              { ident nicht gefunden! }
        else if table[i].kind=proc then
          Error(21);
        GetSym
        end
    else if sym=number then
        GetSym
    else if sym=lparen then
        begin
        GetSym;
        Expression;
        if sym=rparen then
           GetSym
        else
           Error(22)
        end
    else Error(23)
    end; {Factor}

begin {---Term---}                     { Erkennung von Term }
Factor;
while sym in [times,slash] do
    begin
    GetSym;
    Factor
    end
```

```
    end; {Term}

begin {---Expression---}              { Erkennung von Ausdruck}
if sym in [plus,minus] then
    begin
    GetSym;
    Term
    end
else
    Term;
while sym in [plus,minus] do
    begin
    GetSym;
    Term
    end
end; {Expression}

procedure Condition;                  { Erkennen von Bedingung}
begin
if sym=oddsym then
    begin
    GetSym;
    Expression
    end
else
    begin
    Expression;
    if sym in [eql,neq,lss,leq,gtr,geq] then
       begin
       GetSym;
       Expression
       end
    else
```

```
        Error(20)
      end
    end; {Condition}
begin {---Statement---}                   { Erkennen von Anweisung }
if sym=ident then
    begin
    i := Position(id);
    if i=0 then
       Error(11)
    else if table[i].kind <> variable then
      Error(12);
    GetSym;
    if sym=becomes then
       GetSym
    else
       Error(13);
    Expression
    end
else if sym=callsym then
    begin
    GetSym;
    if sym <> ident then
       Error(14)
    else
       begin
       i := Position(id);
       if i=0 then
          Error(11)
       else if table[i].kind <> proc then
          Error(15);
       GetSym
       end
    end
```

```
else if sym=beginsym then
    begin
    GetSym;
    Statement;
    while sym=semicolon do
       begin
       GetSym;
       Statement
       end;
    if sym=endsym then
       GetSym
    else
       Error(17)
    end
else if sym=ifsym then
    begin
    GetSym;
    Condition;


    if sym=thensym then
       GetSym
    else
       Error(16);
    Statement
    end
else if sym=whilesym then
    begin
    GetSym;
    Condition;
    if sym=dosym then
       GetSym
    else
```

```
          Error(18);
        Statement
        end
    end; {Statement}

begin {Block}
ConstDecl;
VarDecl;
while sym=procsym do
   begin
   GetSym;
   ProcDecl(tx)
   end;
Statement
end; {Block}
```

```
{ ================================================================= }
{ Prozedur Init (Initialisierung aller Variablen)                   }
{ ================================================================= }
procedure Init;
var initch: char;
begin
{ -- Initialisierung des Sondersymbol-Arrays ---------------------- }
for initch := chr(0) to chr(CHSETSIZE-1) do
   ssym[initch] := nul;
ssym['+'] := plus;          ssym['-']:= minus;
ssym['*'] := times;         ssym['/']:= slash;
ssym['('] := lparen;        ssym[')']:= rparen;
ssym['='] := eql;           ssym[',']:= comma;
ssym['.'] := period;        ssym['#']:= neq;
ssym['<'] := lss;           ssym['>']:= gtr;
ssym[';'] := semicolon;

{ -- Initialisierung der beiden Schlüsselwort-Arrays -------------- }
word[ 1] := 'begin     ';    word[ 2] := 'call      ';
word[ 3] := 'const     ';    word[ 4] := 'do        ';
word[ 5] := 'end       ';    word[ 6] := 'if        ';
word[ 7] := 'odd       ';    word[ 8] := 'procedure ';
word[ 9] := 'then      ';    word[10] := 'var       ';
word[11] := 'while     ';

wsym[ 1] := beginsym;       wsym[ 2] := callsym;
wsym[ 3] := constsym;       wsym[ 4] := dosym;
wsym[ 5] := endsym;         wsym[ 6] := ifsym;
wsym[ 7] := oddsym;         wsym[ 8] := procsym;
wsym[ 9] := thensym;        wsym[10] := varsym;
wsym[11] := whilesym;

{ -- Initialisierung weiterer Hilfsvariablen ---------------------- }
```

```
cc := 0;
ll := 0;
ch := ' ';
zz := 0;
end; {Init}
```

```
{ ================================================================== }
{ Hauptprogramm für PL/0 Parser                                      }
{ ================================================================== }
begin { main }
writeln('----------------------------------------------------------');
writeln('PL/0 Syntax Analyser [nach Professor Niklaus Wirth]');
writeln('----------------------------------------------------------');
writeln;
Init;                      { Initialisierung der globalen Variablen  }
GetSym;                    { Behandlung des Syntaxdiagrams "program" }
Block(0);                  { Aufruf von Block mit Tabellenindex tx=0 }
if sym = period then
   writeln('Syntaxanalyse ok.')
else
   Error(9)

end.
{*******************************************************************}
{*** End of file pl0.pas *******************************************}
{*******************************************************************}
```

Zu diesem Parser-Programm hier noch ein beispielhafter Programmlauf mit einem kleinen fehlerhaften PL/0-Programm.

```
--------------------------------------------------------------
PL/0 Syntax Analyser [nach Professor Niklaus Wirth]
--------------------------------------------------------------
  const MAX=10;
  var i,j,k;
  begin
  i:=2*MAX;
  j:=i+k/m;
    ^11
```

```
Fehler gefunden in Zeile 5.
Nicht deklarierter Bezeichner: m
```

6.4.5.2. Quelltext des PL/0-Parsers (C-Version)

Hier nun die angekündigte, in ANSI-C formulierte Parser-Version, die bewußt in einzelnen Detailformulierungen von der Pascal-Variante abweicht. Insbesondere werden die teilweise (gegenüber Pascal) komfortableren Möglichkeiten von C genutzt. Auf der anderen Seite musste auf die Blockstruktur der Prozeduren (Funktionen) wegen der Einschränkung von C verzichtet werden.

```
/*
 *  pl0.c
 *
 *  Die Ablaufphilosophie ist hier C-typisch: Im Falle eines Fehlers
 *  wird die terminierende Fehlerroutine aufgerufen. Dadurch entfällt
 *  jeweils der else-Zweig.
 */


/*** Headerdateien ************************************************/
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*** Defines ******************************************************/
#define     NORW        11      /* "number of reserved words        */
#define     TXMAX      100      /* Groesse der Bezeichner-Tabelle   */
#define     NMAX        14      /* maximale Anzahl von Ziffern in   */
                                /* integer-Literalen                */
#define     AL          10      /* akzeptierte Länge von Bezeichnern*/
#define     CHSETSIZE 256       /* 8-Bit-ASCII-Zeichensatz (Groesse)*/
#define     LINELEN     80      /* Zeilenlaenge des Eingabestromes  */

#define     CR          0x0D    /* CarriageReturn                   */
#define     LF          0x0A    /* LineFeed                         */
```

```
/*** Typ-Deklarationen *********************************************/
typedef enum symbol
    {
        NUL, IDENT, NUMBER, PLUS, MINUS, TIMES, SLASH, ODD,
        EQL, NEQ, LSS, LEQ, GTR, GEQ,
        LPAREN, RPAREN, COMMA, SEMICOLON, PERIOD, BECOMES,
        BEGIN, END, IF, THEN, WHILE, DO, CALL,
        CONSTANT, VARIABLE, PROCEDURE
    }
    SYMBOL;

typedef char ALFA[AL+1];  /* +1 wegen ASCII-0 als String-Ende-Marke */

typedef enum object
    {
        CONST, VAR, PROC
    }
    OBJECT;

typedef struct tableentry
    {
        ALFA     name;
        OBJECT   kind;
    }
    TABLEENTRY;

typedef TABLEENTRY TABLE[TXMAX+1];
```

```
/*** Prototypen ****************************************************/
void Error(int);
void GetCh(void);
void GetSym(void);
void Block(void);
void Enter(OBJECT);
int  Position(ALFA);
void ConstDecl(void);
void VarDecl(void);
void ProcDecl(void);
void Statement(void);
void Expression(void);
void Term(void);
void Factor(void);
void Condition(void);

/*** Deklarationen der globalen Variablen *************************/
char    ch=' ';           /* letztes gelesenes Zeichen         */
SYMBOL  sym;              /* letztes gelesenes Symbol          */
ALFA    id;               /* letzter gelesener Bezeichner      */
int     num,              /* letzte gelesene Zahl (num. Wert) */
        cc=0,             /* "character count"                 */
        ll=0,             /* "line length"                     */
        tx=0,             /* Tabellenindex (siehe Var. table) */
        lineno=0;         /* Zeilennummer des Quelltextes      */
char    line[LINELEN];    /* Buffer fuer Eingabezeile          */
ALFA    word[NORW]=       /* Liste der Bezeichner              */
        {
            "begin", "call", "const", "do", "end", "if",
            "odd", "procedure", "then", "var", "while"
        };
```

```
SYMBOL  wsym[NORW]=           /* Liste der entsprechenden Symbole */
        {
            BEGIN, CALL, CONSTANT, DO, END, IF,
            ODD, PROCEDURE, THEN, VARIABLE, WHILE
        },
        ssym[CHSETSIZE]={ NUL };/* Liste der Sondersymbole          */
TABLE   table;                 /* Symboltabelle                    */

/*** Hauptprogramm PL/0-Parser *************************************/
int main(void)
{
    int i;

    /* Initialisierung der globalen Variablen ssym[].              */
    /* Die Arrays word[] und wsym[] wurden schon bei der Deklaration*/
    /* initialisiert.                                              */
    ssym['+']=PLUS;       ssym['-']=MINUS;         ssym['*']=TIMES;
    ssym['/']=SLASH;      ssym['(']=LPAREN;        ssym[')']=RPAREN;
    ssym['=']=EQL;        ssym[',']=COMMA;         ssym['.']=PERIOD;
    ssym['#']=NEQ;        ssym['<']=LSS;           ssym['>']=GTR;
    ssym[';']=SEMICOLON;

    printf("---------------------------------------------------\n");
    printf("PL/0 Syntax Analyser [ANSI-C Version]\n");
    printf("---------------------------------------------------\n\n");
    GetSym();
    Block();
    if (sym!=PERIOD)
        Error(9);
    printf("Syntaxanalyse ok.\n");
    return EXIT_SUCCESS;
} /* end main */
```

```
/*** Error: Fehlermeldung und Programmausstieg *********************/
void Error(int errno)
{
    int i;
    printf("***");
    for (i=0; i<cc; i++)
        putchar(' ');
    printf("^\n*** Fehler #%d aufgetreten in Zeile "
        "%d!\n",errno,lineno);
    printf("Syntaxanalyse abgebrochen.\n");
    exit(EXIT_FAILURE);
} /* end Error */

/*** GetCh: Aktuelles Zeichen bereitstellen ************************/
void GetCh(void)
{
    if (cc==ll) /* d. h. alte Zeile verbraucht, neue Zeile holen */
    {
        if (feof(stdin))
        {
            printf("*** Fehler: Programm unvollständig!\n");
            exit(EXIT_FAILURE);
        }
        cc=0;
        printf("%3d: ",++lineno);
        fgets(line,LINELEN-1,stdin);
        printf("%s",line);
        ll=strlen(line);
    }
    ch=line[cc++];
    if (ch==CR || ch==LF)       /* Umwandlung von CarriageReturn und */
        ch=' ';                 /* LineFeed in Blanks                */
} /* end GetCh */
```

```
/*** GetSym: Aktuelles Symbol bereitstellen ***********************/
void GetSym(void)
{
    while (ch==' ')             /* Leerzeichen werden ueberlesen     */
        GetCh();

    if (isalpha(ch))            /* Buchstaben                        */
    {
        int i=0;
        do
        {
            if (i < AL)         /* Bezeichner a zusammenbauen        */
                id[i++]=ch;
            GetCh();
        } while (isalnum(ch));
        id[i]='\0';             /* Stringende markieren              */

        for (i=0; i<NORW; i++) /* Suche: ist id ein Schluesselwort? */
        {
            if (strcmp(id,word[i])==0)
            {
                sym=wsym[i];
                return;
            }
        }
        sym=IDENT;

    } /* end Buchstaben */
    else if (isdigit(ch))       /* Zifferzeichen                     */
    {                           /* Der numerische Wert des Zahl-     */
        int i=0;                /* literals wird zwar ermittelt u. in*/
        num=0;                  /* der Variablen num gespeichert; er */
        sym=NUMBER;             /* wird jedoch nicht weiter verwendet*/
```

```
    do
    {
        int neuwert;

        if (i>NMAX)         /* Overflow? NMAX=max. Stellenanzahl */
            Error(30);
        neuwert = ch - '0';
        if (num <= (INT_MAX - neuwert) / 10)
            num = 10*num + neuwert;
        else
            Error(31);
        i++;
        GetCh();
    } while (isdigit(ch));
} /* end Zifferzeichen */
else                        /* ab hier: Sonderzeichen           */
{
    switch(ch)
    {
        case ':':   GetCh();
                    if (ch=='=')
                    {
                        sym=BECOMES;
                        GetCh();
                    }
                    else
                        sym=NUL;
                    break;
```

```
            case '<':   GetCh();
                        if (ch=='=')
                        {
                            sym=LEQ;
                            GetCh();
                        }
                        else
                            sym=LSS;
                        break;
            case '>':   GetCh();
                        if (ch=='=')
                        {
                            sym=GEQ;
                            GetCh();
                        }
                        else
                            sym=GTR;
                        break;
            default:    sym=ssym[ch];/* Hiermit wird einem falschen */
                                     /* Zeichen das Symbol NUL zuge-*/
                                     /* wiesen, der eigentlich lexi-*/
                                     /* kalische Fehler somit in die*/
                                     /* syntaktische Phase verlagert*/
                        GetCh();
                        break;
        } /* end switch */
    } /* end Sonderzeichen */
} /* end GetSym */
```

```
/*** Block: Abarbeitung des Syntaxdiagramms Block *****************/
void Block(void)
{
    int prevtx=tx;              /* alten Tabellenindex sichern        */
                                /* Vergleichen Sie dieses Vorgehen mit */
                                /* der Pascal-Implementation in 4.5.1! */
    ConstDecl();
    VarDecl();
    while (sym==PROCEDURE)
    {
       GetSym();
       ProcDecl();
    }
    Statement();

    tx=prevtx;                  /* Tabellenindex tx zuruecksetzen      */
} /* end Block */


/*** Enter: Eintragung in die Tabelle table vornehmen *************/
void Enter(OBJECT obj)          /* Tabelleneintrag: CONST, VAR, PROC   */
{
    tx++;
    strcpy(table[tx].name,id);
    table[tx].kind=obj;
} /* end Enter */


/*** Position: Gueltigkeit eines Bezeichners ueberpruefen **********/
int Position(ALFA id)
{
    int i=tx;
    strcpy(table[0].name,id);
```

```
    while (strcmp(table[i].name,id)!=0)
        i--;
    return(i);              /* entweder Positionsnummer oder 0 fuer*/
} /* end Position */        /* nicht gefunden                     */

/*** ConstDecl: Abarbeitung der entsprechenden Syntax **************/
void ConstDecl(void)
{
  if (sym==CONSTANT)
  {
     do {
         GetSym();
         if (sym!=IDENT)
           Error(4);
         GetSym();
         if (sym!=EQL)
           Error(3);
         GetSym();
         if (sym!=NUMBER)
           Error(2);
         Enter(CONST);
         GetSym();
     } while (sym==COMMA);
     if (sym!=SEMICOLON)
         Error(5);
     GetSym();
  }
} /* end ConstDecl */


/*** VarDecl: Abarbeitung der entsprechenden Syntax ***************/
void VarDecl(void)
{
```

```
  if (sym==VARIABLE)
  {
     do {
         GetSym();
         if (sym!=IDENT)
           Error(4);
         Enter(VAR);
         GetSym();
     } while (sym==COMMA);
     if (sym!=SEMICOLON)
         Error(5);
     GetSym();
  }
} /* end VarDecl */

/*** ProcDecl: Abarbeitung der entsprechenden Syntax ***************/
void ProcDecl(void)
{
    if (sym!=IDENT)
       Error(4);
    Enter(PROC);
    GetSym();
    if (sym!=SEMICOLON)
       Error(5);
    GetSym();
    Block();
    if (sym!=SEMICOLON)
        Error(5);
    GetSym();
} /* end ProcDecl */
```

```
/*** Statement: Abarbeitung des Syntaxdiagramms ********************/
void Statement(void)
{
    int i;
    switch(sym)
    {
        case IDENT:
            i=Position(id);
            if (i==0)
                Error(11);
            if (table[i].kind!=VAR)
                Error(12);
            GetSym();
            if (sym!=BECOMES)
                Error(13);
            GetSym();
            Expression();
            break;
        case CALL:
            GetSym();
            if (sym!=IDENT)
                Error(14);
            i=Position(id);
            if (i==0)
                Error(11);
            if (table[i].kind!=PROC)
                Error(15);
            GetSym();
            break;
```

```
        case BEGIN:
            GetSym();
            Statement();
            while (sym==SEMICOLON)
            {
                GetSym();
                Statement();
            }
            if (sym!=END)
                Error(17);
            GetSym();
            break;
        case IF:
            GetSym();
            Condition();
            if (sym!=THEN)
                Error(16);
            GetSym();
            Statement();
            break;
        case WHILE:
            GetSym();
            Condition();
            if (sym!=DO)
                Error(18);
            GetSym();
            Statement();
            break;
        default:            /* keine Fehlermeldung, da leeres      */
            break;          /* Statement erlaubt ist!              */
    } /* end switch */
} /* end Statement */
```

```
/*** Expression: Abarbeitung des Syntaxdiagramms *******************/
void Expression(void)
{
    if (sym==PLUS || sym==MINUS)
        GetSym();
    Term();
    while (sym==PLUS || sym==MINUS)
    {
        GetSym();
        Term();
    }
} /* end Expression */

/*** Term: Abarbeitung des Syntaxdiagramms *************************/
void Term(void)
{
    Factor();
    while (sym==TIMES || sym==SLASH)
    {
        GetSym();
        Factor();
    }
} /* end Term */
```

```
/*** Factor: Abarbeitung des Syntaxdiagramms ***********************/
void Factor(void)
{
    int i;
    if (sym==IDENT)
    {
        i=Position(id);
        if (i==0)
            Error(11);       /* Identifier nicht gefunden           */
        if (table[i].kind==PROC)
            Error(21);       /* Prozeduren nicht in Factor!         */
        GetSym();
    }
    else if (sym==NUMBER)
        GetSym();
    else if (sym==LPAREN)
    {
        GetSym();
        Expression();
        if (sym!=RPAREN)
            Error(22);
        GetSym();
    }
    else
        Error(23);
} /* end Factor */
```

```
/*** Condition: Abarbeitung des Syntaxdiagramms *******************/
void Condition(void)
{
    if (sym==ODD)
    {
        GetSym();
        Expression();
    }
    else
    {
        Expression();
        if (sym>=EQL && sym<=GEQ)  /* Vergleichsoperatoren          */
        {
            GetSym();
            Expression();
        }
        else
            Error(20);
    }
} /* end Condition */
/******************************************************************/
/*** End of file pl0.c ********************************************/
/******************************************************************/
```

Für den Rahmen unserer Vorlesung soll es hiermit genug des Einblicks in die Theoretische Informatik und den Compilerbau sein. Wer sich jedoch für weitere Aspekte des Compilerbaus interessiert, der sei auf das Buch „Compilerbau" [Baeumle2] hingewiesen.

7. KURZE EINFÜHRUNG: RELATIONALE DATENBANKEN

7.1. Theoretische Grundlagen relationaler Datenbanken

Nachfolgend sollen einige Erläuterungen zum Themengebiet (*Relationale*) *Datenbanken* gegeben werden, die zum Teil den Büchern von Vetter (Aufbau betrieblicher Informationssysteme) [Vetter] und Vossen/Witt (*SQL/DS-Handbuch*) [Vossen] entnommen wurden oder zumindest daran angelehnt sind. Auch das Buch von Schwinn [Schwinn] zu Relationalen Datenbanken wurde für diesen Teil des Skriptums herangezogen. Im Rahmen der FHDW Studienordnung für das Wirtschaftsinformatik-Studium muss allerdings angemerkt werden, dass es eine eigenständige Vorlesung Datenbanken gibt, so dass an dieser Stelle nur einige Grundzüge behandelt werden.

Einen (sowohl praktisch wie auch von den Grundlagen her) interessanten Zugang zu den Beziehungen zwischen den Gebieten Datenbanken, Betriebswirtschaft und Software-Engineering liefert auch das Buch von Wolf Dietrich Nagl: *Computertechnologie und Managementpraxis - Datenbanken und Objekte* [Nagl].

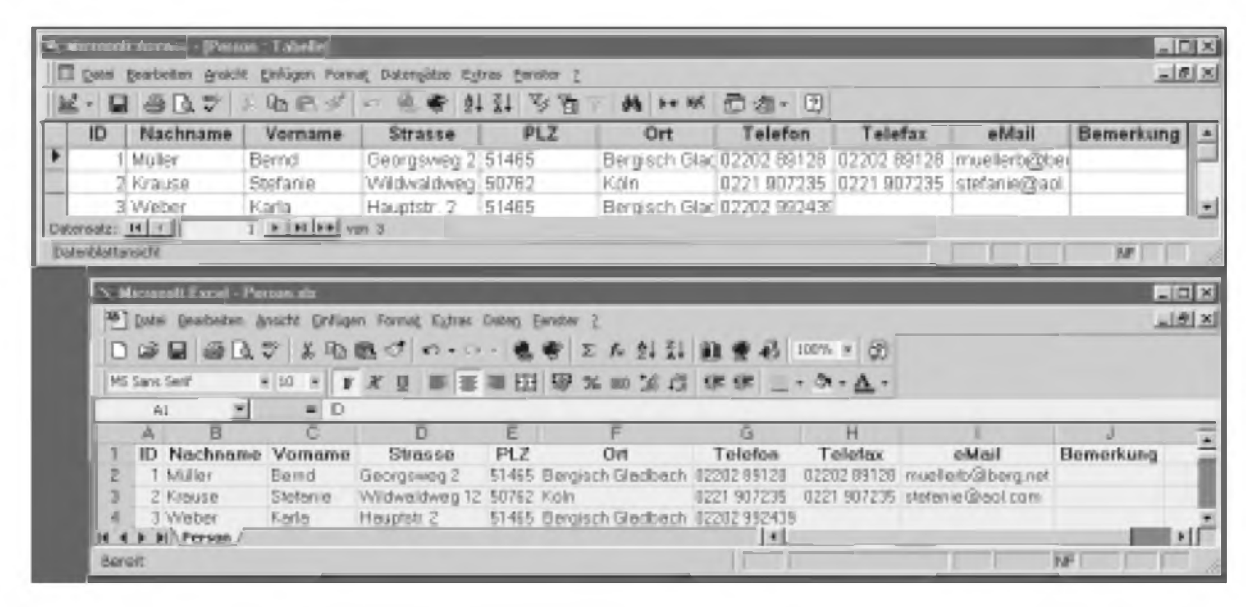


Bild: Ein einfacher Adreßdatenbestand in einer Tabellenkalkulation und in einem Datenbankprogramm

Dabei sollen die im folgenden angestellten theoretischen Betrachtungen auch Antwort auf die Frage geben, warum man nicht „einfach" seine Daten (z.B. Adressen von Lieferanten oder Bestellungen von Kunden) in einer Tabellenkalkulationssoftware verwaltet.

7.1.1. Grundlegende Datenbankmodelle

Eine Datenbank ist eine Ansammlung miteinander in Beziehung stehender Daten, die allen Datenbankbenutzern als gemeinsame Basis aktueller Informationen dient.

Bei den Datenbankmodellen kann unterschieden werden, wie die Identifikation einzelner Datenobjekte realisiert wird: in *hierarchischen* und *Netzwerk-Datenbanken* wird eine referentielle Suche (über in irgendeiner Form codierte Speicherplatzadressen), in *relationalen* Datenbanken eine assoziative (inhaltliche) Suche (nach konkreten Attributwerten) durchgeführt.

Ein relationales Datenbankmanagementsystem (DBMS) definiert sich im wesentlichen durch die folgenden Eigenschaften:

- Die gesamte Information einer relationalen Datenbank wird einheitlich durch Werte repräsentiert, die in Form von (zweidimensionalen) Tabellen dargestellt werden können.
- Der Benutzer sieht keine Verweisstrukturen zwischen diesen Tabellen.
- Es sind (wenigstens) die Operationen zur Selektion (Auswahl) und Projektion (Ausschnitt) sowie zur Verbindung (Join) von Tabellen(einträgen) definiert.

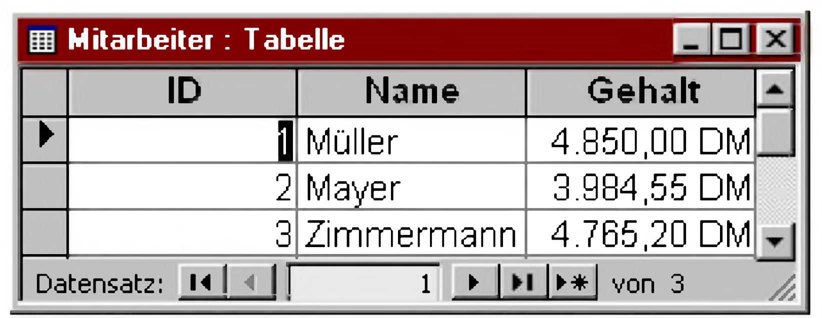
(Zitiert nach [Ortmann], S. 22, bzw. [Codd].)

Diese formale Definition beschreibt natürlich nur das Minimum dessen, was heute für ein relationales Datenbankmanagementsystem üblich ist.

Während sich die obige Definition mit ganz wenigen Kriterien an ein relationales DBMS begnügt, hat Codd 1990 eine Liste mit 333 (!) Kriterien aufgestellt, die ein Datenbanksystem erfüllen muss, um es (wissenschaftlich präzise) relational nennen zu dürfen.

Ein Datenbankmanagementsystem muss jedenfalls stets zwei Grundfunktionen zur Verfügung stellen: die Strukturierung von Datenobjekten und die Repräsentation der (konkreten) Daten.

Diese Grundfunktionen werden gut durch eine Tabellenstruktur umgesetzt: die Spalten einer solchen Tabelle liefern die Struktur (nebenstehend etwa die Felder „ID” für eine fortlaufende eindeutige Nummer, „Name” für den Nachnamen eines Mitarbeiters und „Gehalt” für die monatlichen Bruttobezüge dieses Mitarbeiters.



ID	Name	Gehalt
1	Müller	4.850,00 DM
2	Mayer	3.984,55 DM
3	Zimmermann	4.765,20 DM

Über gemeinsame Felder können nun Beziehungen (Relationen) zwischen verschiedenen Tabellen aufgebaut werden; beispielsweise kann in einer anderen Tabelle die hier gezeigte „ID” (Identifikationsnummer) dazu verwendet werden, in einer Tabelle Mitarbeiteradresse die eindeutige Verbindung zu den hier gezeigten Mitarbeitern (und ihren Gehältern) herzustellen.

Mathematisch wird nun mit den Grundoperationen Selektion und Projektion gearbeitet. Die Selektion wählt (null, eine oder mehrere) bestimmte Zeilen aus, betrifft also die Repräsentation der konkreten Daten. Die Projektion schneidet aus der Datenstruktur die gewünschten Informationen heraus. (Siehe hierzu das kurze „Mathematik-Refreshment" in Abschnitt 7.1.6. auf Seite 208.)

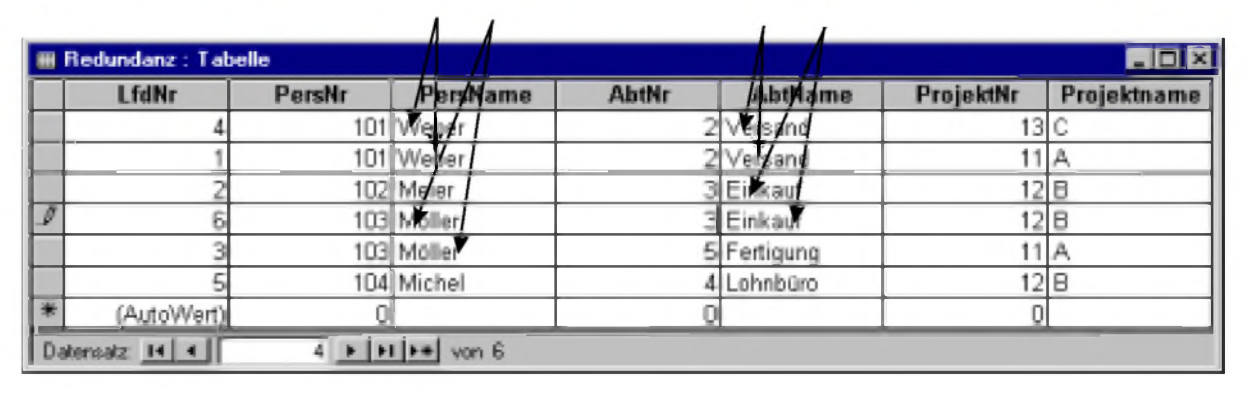
7.1.2. Anforderungen an Datenbanksysteme

Eine relationale Datenbank ist eine Datenbank, in der die Daten in Form von sogenannten Relationen (oder Tabellen) präsentiert werden. Der Zugriff erfolgt auf Basisrelationen, aus denen über entsprechende Verarbeitungsvorschriften gewünschte Ergebnisrelationen (Ergebnistabellen) erstellt werden.

Datenbanksysteme werden eingesetzt zur EDV-gestützten Verwaltung großer Datenbestände. Bei den (von den diversen Betriebssystemen) gewohnten Dateisystemen besteht eine starre Zuordnung von Daten bzw. Dateien zu einzelnen, sie verarbeitenden Programmen; Dateistrukturen müssen dem jeweiligen Anwendungsprogramm angepasst sein, gleiche oder ähnliche Daten sind womöglich (in mehreren Varianten) in verschiedenen Dateien physisch abgespeichert.

Daraus resultieren eine hohe Redundanz (durch Mehrfachabspeicherung von Daten), die Gefahr von Inkonsistenz sowie Inflexibilität gegenüber veränderten Anforderungen der Anwender. Außerdem ist heutzutage bei der Software-Erstellung die Einhaltung von Standards (z.B. hinsichtlich der verwendeten Datenformate) eine übliche Anforderung, die bei der Verwendung üblicher Dateisysteme nicht automatisch erfüllt wird.

Beispiel: Redundanz und inkonsistente Daten treten sehr schnell auf. Die untenstehende Tabelle zeigt neben der Abteilungsnummer (AbtNr) jeweils den Namen der Abteilung (AbtName). Dies ist insoweit redundant als die Nummer der Abteilung eindeutig auf den Namen zurückschließen lässt. Genauso verhält es sich mit PersNr und PersName. Hier könnte etwa zur Abteilungsnummer 2 (eigentlich „Versand") an einer oder mehreren Stellen auch „Fertigung"stehen. Dies wäre ein Fall von Inkonsistenz!



Redundanz : Tabelle

LfdNr	PersNr	PersName	AbtNr	AbtName	ProjektNr	Projektname
4	101	Weber	2	Versand	13	C
1	101	Weber	2	Versand	11	A
2	102	Meier	3	Einkauf	12	B
6	103	Möller	3	Einkauf	12	B
3	103	Möller	5	Fertigung	11	A
5	104	Michel	4	Lohnbüro	12	B
(AutoWert)	0		0		0	

Datensatz: 4 von 6

Und tatsächlich: der Mitarbeiter Möller ist von Abteilung 3 (Einkauf) in die Abteilung 5 (Fertigung) gewechselt (oder umgekehrt); dies ist offenbar bei den hier gezeigten Datensätzen nicht stimmig abgespeichert.

In diesem Sinne haben Datenbanken widerspruchsfrei (bzw. konsistent) und für in der Regel mehrere Benutzer und Benutzerinnen (durch die Vergabe von Zugriffsrechten) gleichzeitig zugänglich zu sein.

Diese Aspekte führen zum Konzept des Datenbanksystems gemäß der nachfolgenden Skizze.

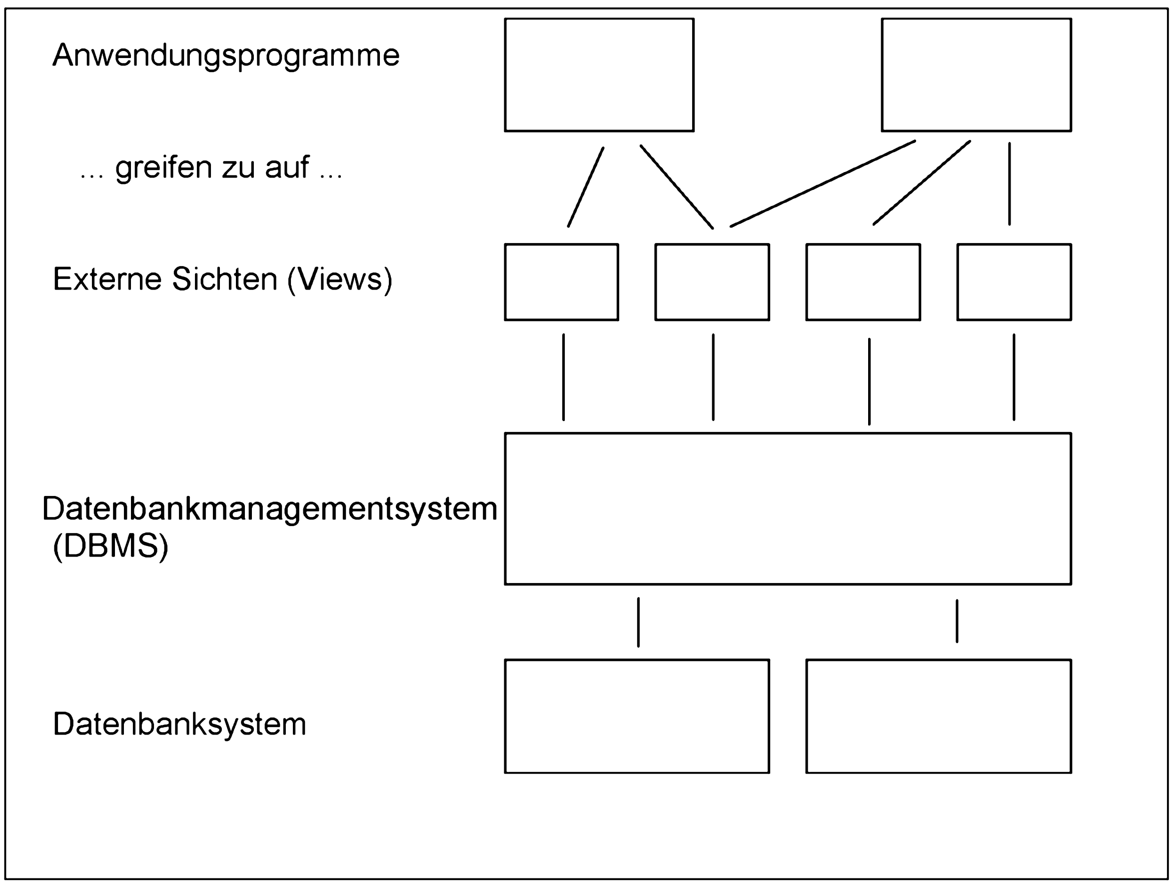


Bild: Grobskizze Datenbanksystem

Die Vorteile eines solchen *Datenbankmanagementsystems* (DBMS): Redundanz wird (weitgehend) vermieden, da Informationen über jedes Objekt (in der Regel) nur einmal abgespeichert und vom DBMS jedem Anwendungsprogramm in entsprechend geeigneter Form zur Verfügung gestellt werden. Soweit Redundanz erwünscht ist, z.B. für effektivere Zugriffe auf die Datenbank, wird sie durch das DBMS kontrolliert, wodurch Inkonsistenzen vermieden werden.

Das DBMS sorgt dafür, dass Programme und Daten getrennt sind, indem es dem Anwendungsprogramm genau die benötigten, angeforderten Datenelemente übergibt; die interne Organisation der Daten bleibt damit (generell) dem zugreifenden Programm verborgen. Diese Trennung von Anwendungsprogramm und Daten wird als *physische Datenunabhängigkeit* bezeichnet.

Es lässt sich zusammenfassen: Eine Datenbank ist eine generell auf Dauer und auf Programmunabhängigkeit ausgelegte Datenorganisation mit einer in der Regel großen Datenmenge und einer dazugehörigen Datenverwaltung.

Datenstrukturierung: Die gesamten Daten sollen einen übersichtlichen inneren Aufbau

haben, so dass ein Anwender sich auf bestimmte Daten oder Datengruppen beziehen kann.

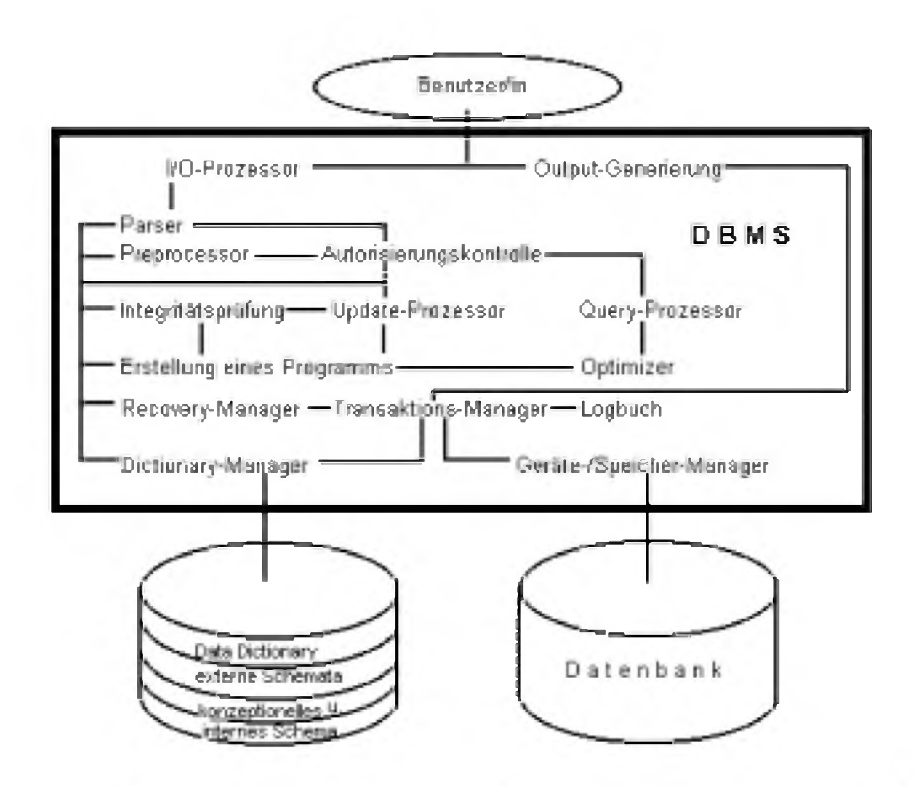
Datentrennung: Durch das Trennen der Daten bzw. der Datenorganisation vom jeweiligen Anwendungsprogramm ist ein unabhängiges Arbeiten möglich. Es können mehrere Anwendungen auf die gleichen Daten zugreifen.

Datenredundanz: Sie entsteht immer dann, wenn gleiche Informationen in mehreren Datensätzen gespeichert werden. Zu verhindern ist dies durch eine saubere Strukturierung der Daten[117].

Datenkonsistenz: Werden in einer Datenbank gleiche Informationen in mehreren Datensätzen gespeichert, so muss gewährleistet sein, dass eine Datenänderung in einem Datensatz ebenso in den anderen Datensätzen vorgenommen wird.

7.1.3. Drei-Ebenen-Architektur

Wichtiges Ziel bei Einführung eines Datenbanksystems ist es, eine solche Datenunabhängigkeit zu erreichen. Dabei wird von der oben erwähnten physischen Datenunabhängigkeit gesprochen, wenn für Programme oder interaktive Datenbankzugriffe die konkrete physische Organisation irrelevant ist. Im Idealfall können dann Daten sogar physisch ganz neu gespeichert werden, ohne dass dies irgendwelche Änderungen an den entsprechenden Anwendungsprogrammen erfordern würde.



Auf der anderen Seite ist es erwünscht, dass ein Datenbanksystem auch logische Datenunabhängigkeit garantiert; es soll unterschieden werden können zwischen einer konkreten,

[117] Es sei allerdings angemerkt, dass bisweilen Redundanz bewußt in Kauf genommen wird um die Performance zu erhöhen. In diesem Fall ist aber ganz besonders darauf zu achten, dass das Datenbanksystem die Konsistenz permanent überwacht.

anwendungsorientierten „Sicht” auf die Datenbank und deren logischer Gesamtstruktur; diese Gesamtstruktur soll verändert werden können, ohne dass alle Sichten geändert werden müssen. Diese Argumente führen zu einem Datenbankdesign auf drei Ebenen gemäß dem *ANSI-Architekturkonzept*.

Die *interne Ebene* ist der physikalischen Abspeicherung am nächsten, ist aber unterscheidbar von dieser, da sie die physischen Daten als interne Datensätze (Records) ansieht. Diese interne Sicht der Daten wird im internen Schema festgelegt: hier werden die Datenstrukturen und erforderlichenfalls spezielle Zugriffsmechanismen definiert.

Auf der *konzeptionellen Ebene* wird die logische Gesamtsicht aller Daten der Datenbank sowie deren Beziehungen untereinander im konzeptionellen Schema (Datenbank-Konzeption) repräsentiert. Die hier verwendeten Datenmodelle abstrahieren von der internen Sicht und der konkreten Abspeicherungsform.

Die *externe Ebene* schließlich beinhaltet die einzelnen Sichten (Views) der einzelnen Benutzerinnen und Benutzer, d.h. der Anwendungsprogramme und der interaktiven Datenbankzugriffe.

Bild: Drei-Ebenen-Architekturkonzept

7.1.4. Komponenten eines Datenbankmanagementsystems

Ein *Datenbanksystem* (DBS), eine *Datenbankumgebung* (*data base environment*, DBE) entsteht durch die Einrichtung einer oder mehrerer (oder formal auch null) Datenbanken unter einem *Datenbankmanagementsystem* (DBMS).

Dabei ist das DBMS die Software, die für die Umsetzung der oben skizzierten Anforderungen und Konzepte zu sorgen hat.

In Erweiterung des Drei-Ebenen-Architekturkonzeptes wird unterschieden zwischen dem DBMS, welches sich als geladenes Programm im Arbeitsspeicher befindet, und den Datenbeständen, auf denen es arbeitet. Einer dieser Datenbestände ist die Datenbank selbst, ein anderer die Beschreibung der den drei Ebenen zugeordneten Schemata, die generell wie die Datenbank selbst auf Massenspeicher abgelegt und als *Data Dictionary* (DD) bezeichnet wird.

Der Benutzer eines Datenbankprogramms nimmt in der Regel nur das sogenannte *Front End* wahr, z.B. die Masken und Dialogfenster von Microsoft Access oder anderen Programmen. Hier werden in irgendeiner Form (Schaltknopf, Texteingabe, Mausklick usw.) Befehle eingegeben („Zeige mir alle Datensätze...”), die das Front End entgegennimmt und zur Bearbeitung an die Kernanwendung weiterreicht.

Ein an die Datenbank gerichteter Benutzerauftrag (in Form eines Kommandos der sogenannten *Data Manipulation Language* (DML) oder eben per Eingabemaske aufbereitet) wird zunächst einem *Parser* übergeben, der die syntaktische Analyse durchführt, also den abgesetzten Befehl auf seine korrekte Syntax (Schreibweise) überprüft. Hierbei werden teilweise auch Informationen aus dem Dictionary benötigt, so dass hier schon eine Verbindung zum sogenannten *Dictionary-Manager* bestehen muss.
Finden diese Aufträge innerhalb von Programmen statt, so wird üblicherweise ein *Precompiler* oder *Preprocessor* erforderlich sein, der eine bestehende Programmiersprache (z.B. *Pascal* oder *C*) um Datenbankzugriffsanweisungen erweitert. Ungeachtet der konkreten Zugriffsform wird ein DB-System jedoch die Zugriffsbefugnis überprüfen müssen: ist ein abgesetzter Befehl für den betreffenden Benutzer überhaupt „legal”?

Bei *Updates* (Aktualisierungen) ist die Ausführung im allgemeinen an *Integritätsbedingungen* geknüpft, welche die Konsistenz der Datenbank gewährleisten sollen. Dabei handelt es sich um Bedingungen der betreffenden Anwendung, beispielsweise „*Gehälter sind stets positiv*” oder „*Kraftfahrzeugkennzeichen sind stets eindeutig*”, welche bei der Definition des konzeptionellen Schemas festgelegt und zur Laufzeit vom DBMS automatisch (ohne Benutzereingriff) überwacht werden. Intern muss in diesem Zusammenhang im Mehrbenutzerbetrieb (Netzwerkumgebungen!) ebenfalls kontrolliert werden, ob sich gegebenenfalls parallel ablaufende Benutzeraufträge beeinflussen.

Weiterhin ist es erforderlich, dass externe Anfragen in das konzeptionelle Schema übersetzt und eventuelle Abkürzungen, die den Anwender(inne)n die Bedienung des Systems erleichtern, durch ihre originären Definitionen ersetzt werden. Dabei muss auch berücksichtigt werden, dass Anfragen unnötig verkompliziert gestellt werden können: ein

Optimizer tut dann sein Bestes, eine (im Hinblick auf Laufzeit und/oder Speicherplatzverhalten) effizientere Formulierung für die gleiche Fragestellung zu finden.

Eine weitere Aufgabe des DBMS besteht in der Erstellung eines Zugriffs- oder Ausführungsprogrammes, einer Code-Generierung für den Benutzerauftrag. Jeder Auftrag, ob programmatisch oder interaktiv abgesetzt, führt schließlich zu einer Sequenz von Lese- und evtl. Schreibbefehlen auf den Speichermedien.

Generell arbeitet nicht ein einzelner Benutzer exklusiv mit einer ganzen Datenbank, sondern diese steht mehreren Personen gleichzeitig offen. Die aus einem einzelnen Auftrag gemäß dem hier beschriebenen Ablauf erzeugte Befehlsfolge wird DBMS-intern als *Transaktion* bezeichnet. Die für den Multi-User-Betrieb zu lösende Schwierigkeit besteht darin, (quasi-)parallel ablaufende Transaktionen zu synchronisieren.

Dieser *Transaktions-Manager* arbeitet nach dem *„Alles-oder-nichts"*-Prinzip: jede Transaktion wird stets *vollständig oder gar nicht* ausgeführt. Bei der Ausführung kann es allerdings vorkommen, dass der Transaktions-Manager feststellt, dass eine gerade laufende Transaktion nicht erfolgreich beendet werden kann; in einem solchen Fall übergibt er sie einem *Recovery-Manager*, dessen Aufgabe es ist, die Datenbank in genau den Zustand zu versetzen, in dem sie sich vor dem Start dieser speziellen Transaktion befunden hat. Dazu wird das *Logbuch* (Protokoll) der Datenbank verwendet, in welchem u.a. die getätigten Veränderungen verzeichnet werden.

Der Recovery-Manager muss ebenfalls aktiv werden, wenn soft- oder hardwaremäßige Fehler im System auftreten; aus diesem Grund ist es in der Praxis erforderlich, dass sich Kopien der Logbücher auf separaten Speichermedien befinden. Zum Teil werden auch mehrere Logbücher parallel (auf verschiedenen Festplatten) mitgeführt.

7.1.5. Der Datenbank-Lebenszyklus

Mit der Planung einer Datenbank beginnt, ähnlich wie beim Software-Engineering für Programme, ein *Datenbank-Lebenszyklus*. Gemäß Vossen/Witt [Vossen] zerfällt dieser in die folgenden Teilschritte (*Top-Down*-Vorgehen), bei denen die ersten vier Schritte unabhängig von der Wahl des konkreten Systems gehalten sind.

7.1.5.1. Anforderungsanalyse und -Spezifikation

Die potentiellen bzw. künftigen Datenbank-Benutzer/innen sind zu befragen, über *was welche* Daten, für *welchen* Zweck *welche* Daten gespeichert werden und *wie* sie verarbeitet werden sollen. Darunter fallen auch Abschätzungen der zu erwartenden Anfrage- und Änderungshäufigkeiten der Daten und die Zuordnung von Zugriffsrechten.

Für die Konsistenz und Integrität der Datenbank ist wichtig festzustellen, welchen Bedingungen (Wertebereiche, Plausibilitäten, Abhängigkeiten) die Daten genügen sollen.

7.1.5.2. Konzeptioneller Entwurf

Der konzeptionelle Entwurf formalisiert das Ergebnis der Anforderungsanalyse. Die Anforderungen einzelner Benutzer(gruppen) werden hierbei zunächst in *externen Sichten* einzeln modelliert und in den *externen Schemata* formal erfasst.

Diese Einzelsichten werden einer Sicht-Integration unterzogen, in eine konzeptionelle Globalsicht der Anwendung umgewandelt; hierbei müssen insbesondere (unerwünschte) Redundanzen und Inkonsistenzen erkannt und beseitigt werden.

Resultat dieses Arbeitsschrittes ist eine vollständige formale Beschreibung der Anwendung, das sogenannte *konzeptionelle Schema*. Die Formalisierung erfolgt mit einem implementierungsunabhängigen, möglichst graphisch darstellbaren Datenmodell, zum Beispiel mit dem *Entity-Relationship-Modell* (ERM).

7.1.5.3. Logischer Entwurf

Im logischen Entwurf wird ein konzeptionelles Schema in ein *logisches Schema* überführt, das ein Datenmodell des zur Verfügung stehenden Datenbankmanagementsystems verwendet, bei uns das Modell der relationalen Datenbanken.

Dieses implementationsunabhängige Datenmodell der Anwendung muss in Beschreibungen der zugrundeliegenden Relationen umgesetzt werden. Jede Relation wird im logischen Schema durch ein *Relationenschema* beschrieben, welches den Namen der Relation, ihre Attributbezeichnungen, die Wertebereiche der Attribute und die Integritätsbedingungen (z.B. in Form von Abhängigkeiten zwischen Attributen) enthält.

Vossen und Witt merken an, dass ein logischer Entwurf vor allem bei kleineren Anwendungen auch ohne vorherigen konzeptionellen Entwurf möglich ist, davon jedoch abzuraten sei, da der konzeptionelle Entwurf implementierungsunabhängig erfolgt und somit durch das Zielsystem bedingte Modellierungseinschränkungen eigentlich zunächst ignoriert werden können.

7.1.5.4. Implementierungsentwurf

Beim *Implementierungsentwurf* wird das logische Schema unter allgemeinen oder aus der Anforderungsanalyse sich ergebenden speziellen Kriterien optimiert. Unter allgemeinen Kriterien ist im Falle des Relationenmodells das *Normalisieren* der Relationen bezüglich ihrer internen Abhängigkeiten zu verstehen, womit Redundanzen vermieden werden.

Die aus dem Normalisierungsprozeß entstehenden Relationen heißen *Basisrelationen*. Zur Realisierung der im konzeptionellen Entwurf festgelegten externen Sichten werden bei der Implementierung die Sichten, denen keine Basisrelationen entsprechen, sondern die sich aus Teilen einer oder mehrerer Basistabellen zusammensetzen, durch sogenannte *Views* beschrieben.

Die Gesamtheit der Relationenschemata für Basisrelationen und Views sowie die vereinbarten Zugriffsrechte bilden das *Datenbankschema* - die Grundlage für die Implementierung.

7.1.5.5. Implementierung

Die *Implementierung* besteht im Einrichten der Datenbank mit der Datendefinitionssprache (engl. *data definition language*, DDL) des konkret zur Verfügung stehenden Datenbankmanagementsystems.

Dies beinhaltet die Implementierung der Basisrelationen und gegebenenfalls der Views, die konkrete Erteilung der Zugriffsrechte und die Festlegung von Indexen als Suchpfaden, über die Anfragen möglichst effizient beantwortet werden können.

Das in SQL implementierte Datenbankschema wird in sogenannten *Systemtabellen* abgelegt.

Bei Microsoft Access stellen sich viele der hier dargestellten Punkte aufgrund der oft suggestiven Windows-Benutzeroberfläche vereinfacht dar; im Hintergrund müssen allerdings die hier aufgeführten Punkte von der Software umgesetzt werden.

7.1.5.6. Arbeiten mit der Datenbank und Reorganisation

Die Arbeit mit einer Datenbank beinhaltet das Einfügen (*insert*), Löschen (*delete*), Ändern (*update*) und Abfragen (*select*) von Daten (mittels der *data manipulation language*, DML oder eben wieder über die Dialogboxen wie bei Access).

Existiert eine Datenbank über einen längeren Zeitraum, so wird sich mitunter nicht nur ihr Inhalt, sondern auch ihre (eigentlich festgelegte) Struktur aufgrund neuer Anforderungen ändern. Dies muss durch die DDL des konkreten Datenbanksystems durch Einrichten, Ändern und Löschen von Relationen, Attributen und Indexeinträgen unterstützt werden.

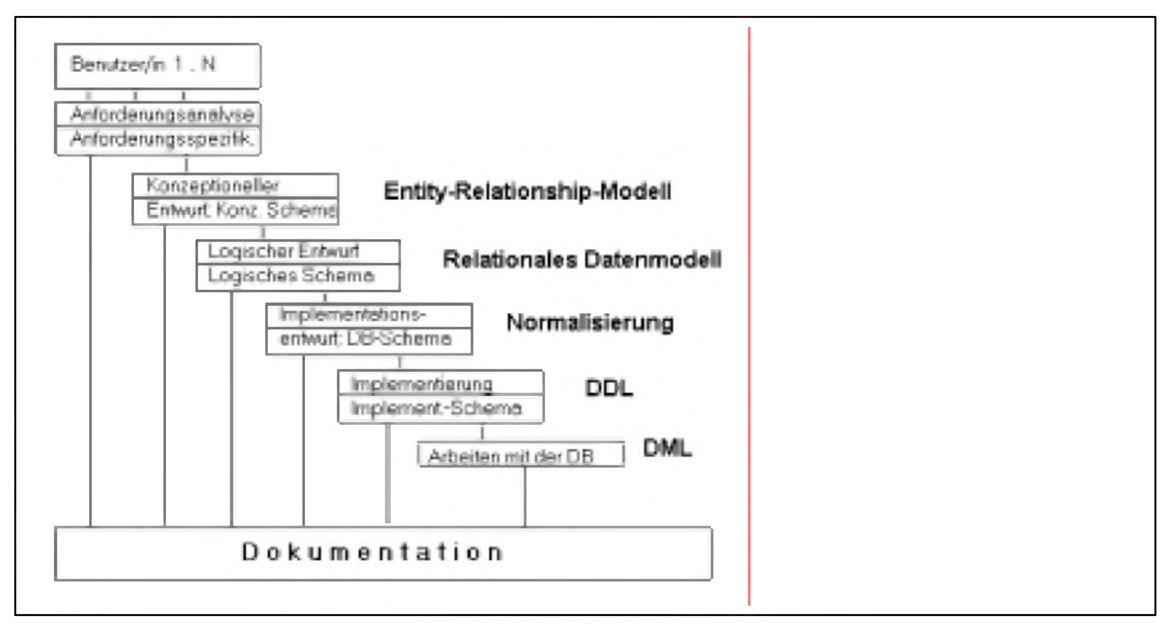


Bild: Datenbank-Lebenszyklus

7.1.6. Zur Wiederholung: Begriffe aus der Mengenlehre

Nachfolgend einige für die Konzeption von Datenbanken relevante Begriffe aus der Mengenlehre; dieser Abschnitt wurde teilweise entnommen aus Vetter [Vetter].

7.1.6.1. Grundlegendes

Eine Menge ist eine eindeutig definierte Zusammenfassung wohlunterscheidbarer Objekte, sogenannter *Elemente*. Sie kann konkret festgelegt werden durch Auflistung aller einzelnen Elemente oder in beschreibender Form. In einem konkreten Kontext ist die Zusammenfassung aller möglichen, zu betrachtenden Objekte die sogenannte *Grundmenge*.

Notation: Eine Menge A kann geschrieben werden mit geschweiften Klammern, z.B. A := { 1,2,3 } oder A := { x | x ist eine natürliche Zahl }.

Eine Menge verfügt (für sich gesehen) über keinerlei Ordnung, keine Reihenfolge. Die Mengen { 1,2,3 } und { 3,1,2 } sind also identisch. Ebenso ist es nur relevant, ob ein Element überhaupt in einer Menge vorkommt oder nicht: { 1,2,3 } und { 1,2,1,3,2 } sind also Beschreibungen für ein und dieselbe Menge (mit drei Elementen).

Notationen: Sei G eine Grundmenge; seien A und B spezielle Mengen, sei x ein Objekt (Element der Grundmenge). Ist x in A enthalten, so schreiben wir $x \in A$ (x ist Element von A); ist dies nicht der Fall, so schreiben wir $x \notin A$ (x ist kein Element von A).

Sind alle Elemente aus A auch in B enthalten, so ist A eine *Teilmenge* von B, B ist eine *Obermenge* von A. (Notationen: $A \subseteq B$ bzw. $B \supseteq A$.)

Ist A eine Teilmenge von B und B eine Teilmenge von A, so sind A und B gleich, andernfalls ungleich. (Das heißt banaler ausgedrückt: A und B sind gleich, wenn sie dieselben Elemente beinhalten.)

Ist kein Element von A in B und kein Element von B in A enthalten, so heißen A und B *disjunkt* (altdeutsch: *elementefremd*).

Sind in B genau all die Elemente der Grundmenge G, die nicht in A enthalten sind, so heißt B das *Komplement* (oder die *Gegenmenge* oder auch *Ergänzungsmenge*) von A. (Manchmal wird das notiert mit $B = G \backslash A$.)

Die Menge ohne jedes Element heißt *leere Menge*, { }.

Hat eine Menge A endlich viele Elemente, so heißt A eine *endliche* Menge, andernfalls spricht man von einer *unendlichen* Menge. Die Anzahl der Elemente heißt auch *Mächtigkeit* der Menge: dies ist eine natürliche Zahl (inklusive der 0) oder das Symbol ∞ (unendlich).

7.1.6.2. Das kartesische Produkt (Geordnete Paare und n-Tupel)

Es seien A und B zwei Mengen, $a \in A$, $b \in B$. Ein *geordnetes Paar* (oder *2-Tupel*) (a,b) ist eine Zusammenfassung der beiden Objekte a und b, bei der die Reihenfolge relevant ist. Das heißt: die geordneten Paare (a,b) und (b,a) sind (falls $a \neq b$ gilt) verschieden voneinander, also $(a,b) \neq (b,a)$.

Das *kartesische Produkt* der Mengen A und B, A x B, ist definiert als die Menge aller geordneten Paare (a,b) mit $a \in A$ und $b \in B$, formelhaft: $A \times B := \{ (a,b) \mid a \in A, b \in B \}$.

Beispiel: Ist A := { 1,2,3 } und B := { 0,1 }, so ist das kartesische Produkt A x B die Menge { (1,0), (1,1), (2,0), (2,1), (3,0), (3,1) }. (Die Mächtigkeit des kartesischen Produktes ist offensichtlich das Produkt der einzelnen Mächtigkeiten von A und von B, was großzügig auch auf das Symbol ∞ ausgedehnt werden kann.) - Aber es kann auch B x B gebildet werden: das ist die Menge { (0,0), (0,1), (1,0), (1,1) }.

In Analogie zum geordneten Paar (oder 2-Tupel) spricht man allgemeiner (für n > 1) von einem (geordneten) *n-Tupel*, wenn n Objekte (unter Beachtung der Reihenfolge) zusammengefasst werden: $(a_1, \ldots, a_n)$.

Sind $A_1, \ldots A_n$ n (beliebige) Mengen (n > 1), so heißt die Menge aller entsprechend gebildeten n-Tupel $(a_1, \ldots, a_n)$ (mit $a_i \in A_i$ für $1 \leq i \leq n$) das kartesische Produkt der Mengen $A_1, \ldots A_n$. (Für n = 3 spricht man von *Tripeln*, bei n = 4 von *Quadrupeln*; die weiteren Werte für n wollen wir hier nervenschonenderweise übergehen.)

Beispiel: Sind V := { DAB1, DAB2, MAT } (Vorlesungen an einer Universität),
P := { Meier, Braunsfeld, Krähwinkel } (Professor(inn)en daselbst) und
R := { Hörsaal-A, Hörsaal-B, Seminarraum }.
Dann ist das kartesische Produkt V x P x R die Menge aller 3-Tupel, die (in dieser Reihenfolge!) aus jeweils einer Vorlesung, einem Professor (oder einer Professorin) und einem Raum bestehen.

Streng formal ist natürlich $V \times P \times R \neq P \times V \times R$; es besteht jedoch eine naheliegende 1:1-Beziehung zwischen diesen beiden Mengen (durch Vertauschen der ersten und zweiten Komponenten).

7.1.6.3. Relationen

Eine *binäre Relation* R (*zweiwertige Relation*) zwischen einer Menge A und einer Menge B ist eine Teilmenge des kartesischen Produktes A x B: $a \in A$ und $b \in B$ stehen in der Relation R zueinander genau dann, wenn $(a,b) \in R$ ist. Eine binäre Relation ist also eine Menge geordneter Paare. Dies kann im Spezialfall auch die leere Menge sein.

Beispiel: Sind V und P die Mengen (Vorlesungen bzw. Professor(inn)en) wie zuvor, dann ist R_2 := { (DAB1,Meier), (DAB2,Braunsfeld), (MAT,Braunsfeld) } eine binäre Relation zwischen V und P. (Eine eventuelle inhaltliche Interpretation ist naheliegend...)

Als Tabelle dargestellt:

R	V	P
	DAB1	Meier
	DAB2	Braunsfeld
	MAT	Braunsfeld

Eine *n-wertige Relation* R (auch *n-stellige Relation* oder *n-ary relation* genannt) zwischen n Mengen A_1 bis A_n (n > 1) ist entsprechend eine Teilmenge des kartesischen Produktes dieser n Mengen (in dieser Reihenfolge), ist somit eine Menge geordneter n-Tupel.

Beispiel: Im obigen Beispiel der Universität ist

R_3 := { (DAB1,Meier,Hörsaal-B),
(DAB2,Braunsfeld,Seminarraum),
(MAT,Braunsfeld,Hörsaal-B),
(DAB1,Krähwinkel,Hörsaal-A) }

also eine dreistellige (oder dreiwertige) Relation.

7.1.6.4. Projektionen

Ist R eine n-stellige Relation zwischen A_1, ... und A_n, dann heißt eine Abbildung (Funktion), die aus jedem n-Tupel von R (bzw. aus A_1 x ... x A_n) eine festgelegte Komponente herausfiltert, eine *elementare Projektion*.

Beispiel: Sei R_3 die dreiwertige Relation von oben. Dann ist die Abbildung

$$p_1 : V \times P \times R \to V$$

mit der Funktionsvorschrift $p_1((v,p,r))$:= v die Projektion auf die 1.Komponente (hier: die Menge der Vorlesungen). So ist zum Beispiel p_1((DAB1,Meier,Hörsaal-B)) = DAB1 und p_1((MAT,Braunsfeld,Hörsaal-B)) = MAT.

Eine Abbildung p, die k der n Komponenten herausfiltert ($k \leq n$), heißt *(allgemeine) Projektion*. Wird eine Projektion auf eine Relation angewendet, so ergibt sich wiederum eine Relation.

Beispiel: Für die obige Relation R_3 ist die Abbildung p_{12} : V x P x R -> V x R mit $p_{12}((v,p,r))$:= (v,p) die Projektion auf die ersten beiden Komponenten. Beispielsweise sind p_{12}((DAB1,Meier,Hörsaal-B)) = (DAB1,Meier) und p_{12}((MAT,Braunsfeld,Hörsaal-B)) = (MAT,Braunsfeld).

7.1.6.5. Natürlicher Verbund (Natural join)

Unter dem *natürlichen Verbund* (*natural join*) versteht man die Verkettung zweier Relationen zu einer neuen Relation. Sind R eine Relation zwischen A und B und S eine Relation zwischen B und C, wobei die Mengen A, B und C selbst bereits kartesische Produkte sein dürfen, dann ergibt sich der natürliche Verbund von R und S als diejenige Relation T, die definiert ist als

$$T := \{ (a,b,c) \mid (a,b) \in R, (b,c) \in S \}.$$

Ein Tupel (a,b,c) liegt somit genau dann im natürlichen Verbund von R und S (also in der Relation T), wenn das Tupel (a,b) in der Relation R liegt und (b,c) zu S gehört.

Beispiel: Es sei R eine binäre Relation zwischen den Mengen A := { S1, S2, S3 } (Matrikelnummern von Student(inn)en) und B := { Herrmann, Müller, Weber } (Namen derselben): R := { (S1,Herrmann), (S2,Müller), (S3,Weber) }. S sei eine binäre Relation zwischen der Menge B und der Menge C := { DAB1, DAB2, MAT }: S := { (Herrmann,DAB1), (Herrmann,MAT), (Weber,DAB2), (Weber,MAT) }. (Der Student Müller ist eine faule Socke.)

Dann ist der natürliche Verbund von R und S die Relation T:= { (S1,Herrmann,DAB1), (S1,Herrmann,MAT), (S3,Weber,DAB2), (S3,Weber,MAT) }.

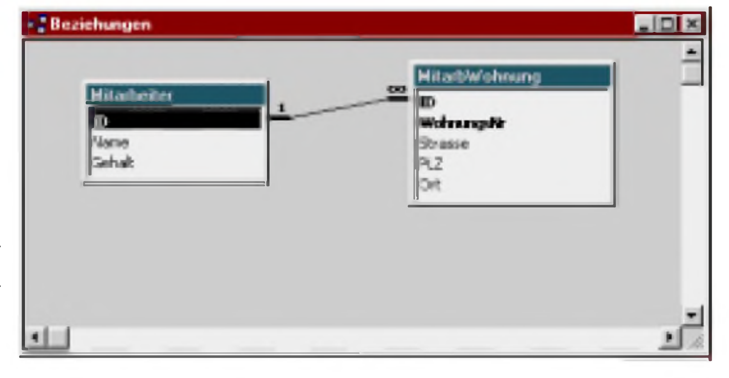
Jedes dieser Dreiertupel hat also die Eigenschaft, dass die ersten beiden Komponenten in Relation R zueinander stehen (-die ersten Komponenten sind die Matrikelnummern der in der zweiten Komponente genannten Studierenden-) und sich die letzten beiden Komponenten in Relation S zueinander befinden (-die Person, die in der 2.Komponente genannt ist, besucht die Vorlesung, die in der 3.Komponente steht-); auf diese Weise ist durch diese natürliche Verknüpfung ersichtlich, welche Matrikelnummern auf die Vorlesungsscheine gehören.

7.2. Datenbankentwurf

Die wichtigsten Themen auf dem Gebiet des Datenbankentwurfes sollen nachstehend angesprochen werden, bevor es im nächsten Kapitel dann um die praktische Arbeit mit dem Datenbanksystem Microsoft Access gehen wird.

7.2.1. Entity-Relationship-Modell

Zur (auch) graphischen Darstellung des konzeptionellen Entwurfs dient das *Entity-Relationship-Modell* (ERM), das die reale Welt in Objekte unterteilt, zwischen denen Beziehungen (*Relationen*) bestehen können. Im nebenstehenden Bild wird eine Beziehung zwischen den Tabellen „Mitarbeiter" und „MitarbWohnung" gezeigt: hier wird modelliert, dass ein Mitarbeiter auch mehrere Wohnungen mit separaten Anschriften haben kann.



Wohlunterscheidbare Dinge, die in der realen Welt existieren, werden *Entities* (unschön eingedeutscht *Entitäten*) genannt. Beispiele hierfür sind „der Kunde mit der Kundennummer 123" oder „das Dienstfahrzeug mit dem Kennzeichen FH-DW 1998".

Einzelne Entities, welche in einem speziellen Sinn ähnlich oder zusammengehörig sind, können zu *Entity-Sets* zusammengefasst werden. Beispiele hierfür sind „alle Mitarbeiterinnen und Mitarbeiter der Fabrik X" oder „alle Dienstfahrzeuge der Firma Y".

Entities besitzen Eigenschaften (etwa Farbe, Länge, Größe, Geburtsdatum, Anschrift etc.), die *Werte* genannt werden. Die Zusammenfassung aller möglichen oder aller zugelassenen Werte für eine Eigenschaft nennt man deren *Wertebereich*. Auf der Ebene eines Entity-Sets werden die bei allen Entities (dieses Sets) auftretenden Eigenschaften *Attribute* genannt.

Beispiel: In einer *FHDW-Studentenverwaltung* können beispielsweise Informationen über Student/inn/en, Dozent/inn/en und den konkreten Vorlesungsplan festgehalten werden. Entities sind hier die Studentin Meier [sofern dies bereits eine Person eindeutig festlegt!], Dozent Prof. Schäfer oder die Vorlesung „Wirtschaftsmathematik I" in der Gruppe „Mittelständische Wirtschaft MW91".

Die Student/inn/en bilden das Entity-Set Student, die Dozent/inn/en das Entity-Set Dozent, die konkret (z.B. in einem bestimmten Semester) geplanten und gehaltenen Vorlesungen das Set Vorlesung. - Attribute mit möglichen Wertebereichen in diesem Zusammenhang können z.B. sein:

Nachname	Characterstring (Zeichenkette)
Bemerkung	Characterstring mit 256 Zeichen
MitarbeiterNr	achtstellige ganze Zahlen
Gehalt	numerisch mit zwei Nachkommastellen
Telefonkurzwahl	numerisch
Geschlecht	Character (einzelnes Zeichen)

Es kann erforderlich sein, für gewisse Attribute neben den eigentlich möglichen, denkbaren Werten einen sogenannten *Nullwert* (ein englisch ausgesprochenes *NULL*) zuzulassen. Dieser Nullwert kann dann verwendet werden, falls ein konkreter Wert für ein Entity dieses Sets noch nicht bekannt ist oder vielleicht nie existieren wird. So kann z.B. das numerische Attribut Telefonkurzwahl gedanklich „leer" bleiben, wenn für eine/n Mitarbeiter/in eine solche Kurzwahl (noch) nicht existiert, datenbanktechnisch wird jedoch durch den besonderen Wert NULL eine Abfrage auf diesen Wert ermöglicht und es kann festgestellt werden, dass eben keine Kurzwahl eingetragen ist.
Wichtig: Dieser Nullwert hat mit der ansonsten auftauchenden Zahl 0 nichts zu tun!

Andere Attribute wiederum dürfen vom Verwendungszweck her nicht „leer" bleiben, dort wird man davon ausgehen müssen, dass stets ein Wert zu jedem Entity eingetragen ist; dies ist z.B. bei der MitarbeiterNr denkbar. Dieses Attribut darf also keine Nullwerte aufnehmen[118], es ist im Gegenteil (in der Praxis) ein identifizierendes Attribut, ein sogenannter *Schlüssel* (*key*), d.h. etwas mit einer minimalen identifizierenden Eigenschaft. Konkret wird über das Schlüsselattribut MitarbeiterNr ein/e konkrete/r Mitarbeiter/in eindeutig zu finden sein.

Die Beziehungen unter Entity-Sets heißen auch *Relationships*. In deren Deklarationen werden die (prinzipiell) zeitunabhängigen Aspekte festgelegt, konkrete Tupel dieser Relationen stellen dann die zeitveränderlichen Informationen dar.

Beispiel: Das Entity-Set Person beinhalte verschiedene Attribute über Personen, z.B. Name, Vorname, Geburtsdatum etc. Eine Relation Mutter über den (beiden) Entity-Sets Person und Person (!) kann dann für konkrete Tupel ausdrücken, dass eine Person die Mutter einer anderen ist. Das spezielle Tupel Mutter(Helga,Ortwin) ist (wegen des männlichen Vornamens, der im Tupel auftritt) semantisch eindeutig als „Helga ist die Mutter von Ortwin" interpretierbar. Das Tupel Mutter(Helga,Anna) zeigt jedoch, dass solche semantischen Interpretationen nicht immer für die gewünschte Information ausreichen, d.h. de facto spielt die Reihenfolge in einer Relation eine gravierende Rolle, die anfangs natürlich definiert werden muss. (So wird im obigen Kontext Mutter(Helga,Anna) sinnvollerweise für „Helga ist die Mutter von Anna" stehen.)

---

[118] Hier muss man dann bei Datenbankprogrammen wie Access die Auswahl „Eingabe erforderlich" ankreuzen, damit die Software überwacht, dass das betreffende Feld auch immer ausgefüllt wird.

Beispiel: Es sei wieder ein Entity-Set Person gegeben mit den Attributen PersNr (als Schlüssel), Nachname, Vorname. Konkrete Tupel könnten die Folgenden sein.

PersNr	Nachname	Vorname
100	Berg	Gustav
101	Berg	Theodor
102	Berg	Ulrike
103	Berg	Viktor
104	Berg-Feuerstein	Andrea

Eine Relation Verheiratet auf den Sets Person und Person kann nun Tupel der Gestalt Verheiratet(100, Berg, Gustav, 102, Berg, Ulrike) besitzen. Im Zusammenhang der gesamten Datenbank kann aber auch mit der viel „kleineren" Relation Verh mit Tupeln der Gestalt Verh(100,102) dieselbe Information verwaltet werden, da die beiden verwendeten Attribute der Relation Verh gerade die Schlüsselattribute der beiden beteiligten Entity-Sets sind.

In der graphischen ERM-Darstellung werden Entity-Sets rechteckig dargestellt, Relationen mit Rauten und die beteiligten Attribute in Ellipsen jeweils angehängt; Schlüsselattribute werden unterstrichen. Dabei können Attribute auch an Relationen angehängt werden, wie es bei der Relation Verheiratet der Fall ist. Da in der Relation jedoch mindestens die Schlüsselattribute der beteiligten Entity-Sets vorkommen müssen, werden diese oftmals in der Graphik weggelassen.

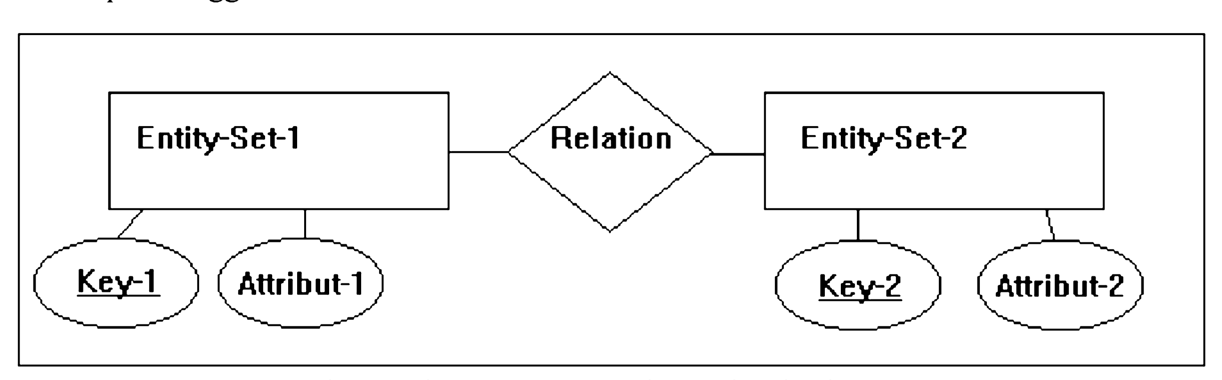


Bild: Graphische Darstellung - Entity-Relationship-Diagramm

Der am häufigsten auftretende Fall in der Praxis sind zweistellige Relationen, d.h. Relationen auf zwei (nicht unbedingt verschiedenen) Entity-Sets. Unter der *Komplexität* einer solchen Relation versteht man die Angabe darüber, mit wievielen Entities des ersten Sets ein bestimmtes Entity des zweiten Sets in Beziehung stehen kann, darf, evtl. im Zusammenhang auch muss. (Anmerkung: Den Begriff der Komplexität gibt es auch allgemein für mehrstellige Relationen.)

Die einfachste Komplexität einer zweistelligen Relation ist 1:1 (*one-to-one*), bei der jedes Entity aus dem einen Set mit höchstens (!) einem Entity des zweiten Sets in Beziehung steht (und umgekehrt). Ein Beispiel hierfür ist im abendländischen Kulturraum die Relation Verheiratet.

Eine Relation zwischen den Entity-Sets E_1 und E_2, bei welchem ein Entity in E_1 mit keinem, einem oder mehreren Entities von E_2 in Beziehung steht, jedes Entity in E_2 sich jedoch höchstens mit einem Entity aus E_1 in Relation befindet, wird *many-to-one* (m:1 oder 1:n) genannt. Ein Beispiel hierfür sind die Relationen Mutter oder Vater.

Bestehen diese beiden Einschränkungen nicht, so spricht man von einer m:n-Beziehung (*many-to-many*). Hierfür ist ein Beispiel eine Relation Unterrichtet(Dozent,Fach), mit der ausgedrückt werden kann, dass ein Dozent mehrere Fächer unterrichten kann, ein Fach aber auch (in der Regel) von mehreren Dozenten unterrichtet wird.

Eine besondere Relation hört auf den Namen *IS-A*. Dabei handelt es sich um eine zweistellige Relation zwischen Entity-Sets E_1 und E_2 („E_1 ist ein E_2"), mit der ausgedrückt wird, dass es sich bei E_1 um eine Spezialisierung von E_2 handelt. Dies ist im Prinzip dasselbe wie eine Teilmengenbeziehung, d.h. E_1 könnte auch in E_2 aufgenommen, integriert werden. Die entsprechend hinzukommenden Attribute von E_1 würden in diesem Fall jedoch bei sämtlichen Tupeln von E_2 (mit sinnvollerweise NULL-Werten) mit auftreten, was weder realisierungstechnisch noch logisch wünschenswert scheint.

Beispielsweise kann ein Entity-Set Abteilungsleiter in eine IS-A-Beziehung zum Set aller Mitarbeiter gebracht werden; neben den allgemeinen, auch für Abteilungsleiter zutreffenden Attributen, die für Mitarbeiter/innen verwaltet werden, können in das Set Abteilungsleiter alle spezifisch zutreffenden Attribute aufgenommen werden.

Der Fall einer 1:n-Beziehung kann (technisch) ohne eigenständige Relation abgespeichert werden. Eine Relation „Wird_Erteilt_Von" zwischen den Entity-Sets Auftrag und Kunde kann mit in das Set Auftrag integriert werden, indem der Schlüssel KundenNr aus Kunde als eigenes Attribut zu Auftrag hinzugefügt wird.

7.2.2. Normalisierung

Ziel eines Datenbanksystems ist es unter anderem, zu jedem Zeitpunkt widerspruchsfreie, konsistente Datenbestände zu verwalten. Fehlen oder widersprechen sich Informationen, so bezeichnet man dies als Speicheranomalie. Um diese zu vermeiden, somit die Konsistenz zu erhalten, ist die Normalisierung der Relationen (Tabellen) erforderlich, die in folgende Stufen eingeteilt wird.

7.2.2.1. Unnormalisierte Relation

Eine unnormalisierte Relation ist daran erkennbar, dass sie Attribute (Spalten) mit mehrelementigen Attributswerten beinhalten kann.
Das Hauptproblem bei unnormalisierten Tabellen ist die Tatsache, dass es EDV-technisch aufwendig(er) ist, einen Datenbestand zu verwalten, bei dem in jeder Zeile (in jedem Tupel) verschieden viele Eintragungen auftreten können.

Beispiel: Die folgende Tabelle stellt eine unnormalisierte Relation dar, denn das Attribut Studienfach kann mehrere Attributwerte aufweisen.

StudentNr	Name	Studienfach
S1001	Meier, Gerda	Mathematik, Physik
S1002	Müller, Erwin	Mathematik, Chemie
S1003	Weber, Karla	Jura

7.2.2.2. Erste Normalform (1NF)

Eine Relation ist in erster Normalform (1NF), wenn alle nicht dem Schlüssel zugehörigen Attribute funktional abhängig vom Gesamtschlüssel sind; anders ausgedrückt: in einer 1NF-Relation hat jedes Attribut stets höchstens einen Wert.

Beispiel: Die folgende Tabelle stellt eine Relation in erster Normalform dar.

StudentNr	Name	StudFachNr	Studienfach
S1001	Meier, Gerda	1	Mathematik
S1001	Meier, Gerda	2	Physik
S1002	Müller, Erwin	1	Mathematik
S1002	Müller, Erwin	2	Chemie
S1003	Weber, Karla	1	Jura

7.2.2.3. Zweite Normalform (2NF)

Ein Problem, das in dem Beispiel eben bei einer 1NF-Relation auftritt, ist die prinzipielle Möglichkeit, einen Eintrag wie z.B. „S1003, Huber, Florian ...” in die Datenbank aufzunehmen, da der Schlüssel zusammengesetzt ist aus StudentNr und StudFachNr. Gleichzeitig erkennen wir jedoch, dass das Attribut Name funktional abhängig ist bereits vom Schlüsselteil StudentNr alleine! Dies führt zu der folgenden Definition der zweiten Normalform.

Eine Relation befindet sich in zweiter Normalform (2NF), wenn jedes nicht zum Schlüssel gehörige Attribut funktional abhängig ist vom Gesamtschlüssel, nicht aber von Schlüsselteilen. Anders formuliert: eine Relation ist 2NF, wenn sie 1NF ist und alle nicht zum Schlüssel gehörenden Attribute voll funktional abhängig sind vom Schlüssel.

Anmerkung: Somit kann die zweite Normalform nur dann verletzt werden, wenn eine Relation einen zusammengesetzten Schlüssel und (mindestens) ein weiteres, nicht zum Schlüssel gehörendes Attribut aufweist.

Um die Tabelle aus dem Beispiel in zweite Normalform zu überführen, ist eine Aufspaltung in zwei einzelne Relationen, wie im folgenden Beispiel gezeigt, erforderlich.

Beispiel: Die folgenden Tabellen stellen Relationen in zweiter Normalform dar.

StudentNr	StudFachNr	Studienfach
S1001	1	Mathematik
S1001	2	Physik
S1002	1	Mathematik
S1002	2	Chemie
S1003	1	Jura

StudentNr	Name
S1001	Meier, Gerda
S1002	Müller, Erwin
S1003	Weber, Karla

7.2.2.4. Dritte Normalform (3NF)

Betrachten wir die folgende Tabelle.

StudentNr	StudFachNr	Studienfach	Fakultät
S1001	1	Mathematik	Naturwissenschaften
S1001	2	Physik	Naturwissenschaften
S1002	1	Mathematik	Naturwissenschaften
S1002	2	Chemie	Naturwissenschaften
S1003	1	Jura	Sonstige Wissenschaften
S1003	2	Mathematik	Naturwissenschaften

Dann sind offensichtlich die Voraussetzungen der zweiten Normalform erfüllt. (Klar?) Gleichzeitig fällt auf, dass ein Eintrag der Form (S1004,1,Jura,Naturwissenschaften) theoretisch möglich wäre, obwohl wir inhaltlich feststellen können, dass dieser Eintrag nicht korrekt wäre, also semantische Inkonsistenz hervorrufen würde!

Grund dafür ist, dass das Nicht-Schlüssel-Attribut Fakultät funktional abhängig ist vom Nicht-Schlüssel-Attribut Studienfach: es ist beispielsweise klar, dass die Werte Mathematik, Physik und Chemie zur Fakultät Naturwissenschaften gehören.

Um auch diesen Mißstand auszuschließen, wird die dritte Normalform wie folgt definiert.

Eine Relation ist in dritter Normalform (3NF), wenn jedes nicht dem Schlüssel angehörende Attribut voll funktional abhängig ist vom (Gesamt-)Schlüssel und es keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Attributen gibt. Anders formuliert: eine Relation ist 3NF, wenn sie 2NF ist und keine sogenannten transitiven Abhängigkeiten besitzt.

Beispiel: Aus obiger Tabelle werden durch Überführung in die dritte Normalform somit die folgenden beiden Tabellen.

StudentNr	StudFachNr	Studienfach
S1001	1	Mathematik
S1001	2	Physik
S1002	1	Mathematik
S1002	2	Chemie
S1003	1	Jura
S1003	2	Mathematik

Studienfach	Fakultät
Mathematik	Naturwissenschaften
Physik	Naturwissenschaften
Chemie	Naturwissenschaften
Jura	Sonstige Wissenschaften

In der Praxis weniger relevant, theoretisch jedoch ebenfalls denkbar, sind weitere Problemsituationen, die die vierte bzw. fünfte Normalform erforderlich machen (können). Der Vollständigkeit halber soll daher kurz auf die 4NF und die 5NF eingegangen werden, auch wenn man sich in der Praxis mit der dritten Normalform begnügt.

7.2.2.5. Vierte Normalform (4NF)

Beispiel: Betrachten wir die folgenden beiden Relationen Spricht und Produziert.

Spricht

PERSONNR	SPRACHE
101	DEU
101	ENG
102	DEU
102	FRZ

Produziert

PERSONNR	PRODUKTNR
101	11
101	12
102	11
102	13

Über diese beiden Relationen ist das Attribut PersonNr zweimal komplex assoziiert, d.h. zu einer PersonNr gibt es jeweils mehrere Attributwerte von Sprache bzw. von ProduktNr.

Über den natürlichen Verbund (*natural join*) dieser beiden Tabellen gelangen wir zu einer Relation Person, die in der folgenden Aufstellung gezeigt wird.

Person

PERSONNR	SPRACHE	PRODUKTNR
101	DEU	11
101	DEU	12
101	ENG	11
101	ENG	12
102	DEU	11
102	FRZ	11
102	DEU	13
102	FRZ	13

Beachten Sie bitte, dass eines dieser Tupel nur durch Angabe aller drei Attribute herausgefiltert werden kann, also alle drei Attribute Schlüsselbestandteile sein müssen, wodurch automatisch auch die dritte Normalform sichergestellt ist.

Weiterhin ist festzustellen, dass für jede von einer Person (z.B. 101) gesprochene Sprache eine identische Menge von ProduktNr-Werten (hier 11 und 12) erscheint. Trifft eine solche Bedingung für jede Person zu, so spricht man davon, dass das Attribut ProduktNr mehrwertig abhängig (oder mengenwertig abhängig) ist vom Attribut PersonNr.

Entsprechend entdecken wir eine weitere mehrwertige Abhängigkeit: für jedes von einer bestimmten Person produzierte Produkt erscheint stets dieselbe Menge gesprochener Sprachen. Und so ist das stets: mehrwertige Abhängigkeiten erscheinen in einer Relation immer paarweise!

Angenommen, Person 101 produziert nicht länger das Produkt Nr. 11, dann müssten die beiden Tupel (101,DEU,11) und (101,ENG,11) gelöscht werden. Aus der Relation Person kann aber, weil alle drei Attribute gemeinsam den Schlüssel bilden, ein einzelnes Tupel gelöscht werden, womit Person (inkonsistenterweise) nicht mehr länger die komplexen Assoziationen zwischen PersonNr, Sprache und ProduktNr ausdrücken würde. Ähnlich verhielt es sich bei dem Problem, neue Daten einzufügen, beispielsweise wenn Person 102 zusätzlich Englisch lernt und notwendigerweise die beiden Tupel (102,ENG,11) und (102,ENG,13) eingefügt werden müssten.

Diese Problembetrachtung führt zur folgenden Definition: Eine Relation ist in vierter Normalform (4NF), wenn sie 3NF ist und keine mehrwertigen Abhängigkeiten besitzt.

7.2.2.6. Fünfte Normalform (5NF)

Beispiel: Betrachten wir die modifizierte, aus obiger Relation Person abgewandelte Relation Person2.

Person2

PERSONNR	SPRACHE	PRODUKTNR
101	DEU	11
101	DEU	12
101	ENG	11
101	ENG	12
102	DEU	11
102	FRZ	11
102	DEU	13

Gegenüber der Relation Person wurde also lediglich das Tupel (102,FRZ,13) entfernt. Damit ist erkennbar, dass mit einer Zerlegung der Relation Person2 in die zwei durch Projektionen (vgl. Abschnitt 7.1.6.4. auf Seite 210 auf jeweils zwei Spalten entstandenen Relationen Spricht(PersonNr,Sprache) und Produziert(PersonNr,ProduktNr) die ursprüngliche Relation Person2 nicht mehr rekonstruiert werden kann. Dies liegt daran, dass der natürliche Verbund der Relationen Spricht und Produziert zu einem Tupel führt, das in Person2 nicht (mehr) enthalten ist.

Wird dagegen die Relation Person2 in die drei Projektionsrelationen Spricht(PersonNr, Sprache), Produziert(PersonNr, ProduktNr) und Spr-Prod(Sprache, ProduktNr) zerlegt, so ist (über einen zweistufigen natürlichen Verbund) die Ausgangsrelation Person2 wieder eindeutig rekonstruierbar.

```
PERSON2                                    Illustration zur 5.Normalform
101, DEU, 11
101, DEU, 12
101, ENG, 11
101, ENG, 12
102, DEU, 11
102, FRZ, 11
102, DEU, 13

Projektionen führen zu den Relationen SPRICHT, PRODUZIERT und SPR-PROD ...

SPRICHT              PRODUZIERT             SPR-PROD
101,DEU              101, 11                DEU, 11
101,ENG              101, 12                DEU, 12
102,DEU              102, 11                ENG, 11
102,FRZ              102, 13                ENG, 12
                                            DEU, 13
                                            FRZ, 11

Der natürliche Verbund von SPRICHT und PRODUZIERT führt zur Relation PERSON ...

PERSON
101, DEU, 11
101, DEU, 12
101, ENG, 11
101, ENG, 12
102, DEU, 11
102, DEU, 13
102, FRZ, 11
102, FRZ, 13


Der natürliche Verbund von PERSON und SPR-PROD schließlich führt zu PERSON2 ...

PERSON2
101, DEU, 11
101, DEU, 12
101, ENG, 11
101, ENG, 12
102, DEU, 11
102, FRZ, 11
102, DEU, 13
```

Bild: Illustrierendes Beispiel zur 5.Normalform

Das hier aufgeführte Beispiel der Relation Person2 kann (prinzipiell) in der Praxis zu Speicheranomalien führen, so dass es (theoretisch) einleuchtend erscheint, auch die fünfte Normalform (nach [Vetter]) zu definieren:

Eine Relation ist in fünfter Normalform (5NF), wenn sie unter keinen Umständen aufgrund einer Verschmelzung einfacherer, d.h. weniger Attribute aufweisender, Relationen mit unterschiedlichen Schlüsseln konstruiert werden kann.

Allerdings geben auch diejenigen Autoren, die die 5NF erstmals definiert haben, zu, dass es in der Praxis häufig nur sehr schwer herauszufinden ist, ob eine Relation die 5NF erfüllt oder nicht. Generell wird davon ausgegangen, dass (3NF- oder 4NF-)Relationen, die die 5NF

verletzen, eher konstruierte Beispiele sind und in der Praxis nur sehr selten vorkommen. Dieser Hoffnung wollen wir uns im Folgenden einfach anschließen.

7.2.3. Integrität

Um einen Datenbestand widerspruchsfrei zu halten, ist ebenfalls auf die sogenannte *Integrität* zu achten. Man unterscheidet dabei häufig drei Arten von Integrität:

- Entity-Integrität
- Referentielle Integrität
- Benutzerdefinierte Integrität (Semantische Integrität)

Dabei sollte in der Praxis das Datenbank-Managementsystem dafür sorgen, dass weitmöglichst alle Integritäten automatisch überwacht werden; wo dies nicht möglich oder zu umständlich ist, müssen eigene Zugriffsprozeduren (Eingabe-, Lösch- und Veränderungsroutinen) für die Einhaltung der Integritätsbedingungen sorgen.

7.2.3.1. Entity-Integrität

Entity-Integrität bedeutet, dass die Primärschlüssel keine NULL-Werte annehmen können bzw. dürfen. Dies entspricht der Tatsache, dass Entities wohlunterscheidbare Objekte der realen Welt darstellen, also insbesondere eindeutig identifizierbar sein müssen.

7.2.3.2. Referentielle Integrität

Korrespondiert ein Fremdschlüssel X einer Relation R1 mit dem Primärschlüssel Y der Relation R2, so muss jeder Wert von X in R1 auch in Y bei R2 vorkommen oder er muss (insgesamt) ein NULL-Wert sein. Mit anderen, vielleicht verständlicheren Worten: ist bei einem Fremdschlüssel ein Eintrag vorhanden, so muss der Bezug, der dadurch hergestellt wird, in einer anderen Relation auch erfüllt bzw. aufgelöst werden.

Beispiel: Taucht in einer Relation R1 zu einem Angestellten als Fremdschlüssel die Mitarbeiternummer seines Vorgesetzten auf, so muss es natürlich zwingend die dort eingetragene Mitarbeiternummer (des Vorgesetzten) entweder im Mitarbeiterstamm auch geben, oder aber bei der Mitarbeiternummer des Vorgesetzten steht ein NULL-Wert, d.h. der Angestellte hat (derzeit) keinen (direkten) Vorgesetzten.

7.2.3.3. Benutzerdefinierte Integrität (Semantische Integrität)

Neben den o.g. Integritäten kann es auch noch weitere, inhaltliche Bedingungen geben, die jederzeit erfüllt sein müssen, damit der Datenbestand konsistent und sinnvoll ist. So kann es für den konkreten Betrieb z.B. festgelegt sein, dass bei den Mitarbeiternummern die Angestellten mit einer führenden 1, die Arbeiter/innen mit einer führenden 2 und die Leitenden Angestellten mit einer führenden 0 (!) gekennzeichnet werden. Diese Vorgabe auch einzuhalten, wäre ein Beispiel einer benutzerdefinierten oder semantischen Integrität.

7.2.4. Beispiel zum Datenbankentwurf: die Datenbank Auftrag

Im Folgenden werden wir anhand des Beispiels der Datenbank *Auftrag* die Schritte, die beim Datenbankentwurf notwendig sind, durcharbeiten.

Nehmen wir an, Sie speichern alle Kunden-, Bestell- und Artikeldaten in einer einzigen Tabelle[119], so müssen Sie bei Bestellungen von unterschiedlichen Kunden jeweils alle Bestell- und Artikeldaten nochmals erfassen. Bestellt dagegen ein Kunde mehrmals, so wiederholt sich zwangsläufig die Eingabe aller kundenspezifischen Daten.

In der folgenden Abbildung wiederholen sich so die Bestellungen der Artikel mit den Artikelnummern 5534 (Damenhose) und 6643 (Sweatshirt) und die Angaben z.B. zu dem Kunden „A & B". (Abgesehen davon hat dieser Kunde zwei verschiedene Kundennummern erhalten, aber das ist ein anderes Problem.)



Kunde : Tabelle

KdNr	Firma	Adresse	Ansprechpartner	TelNr	BestN	ArtNr	ArtName	Menge
101	Meierdinck	Waldweg 3, 50736 Köln	Gabriele Meierdinck	0221 / 76788	1302	5534	Damenhose	100
254	A & B	Bahnhofstr. 2, 51465 Bergisch Gladbach	Hans Meischner	02202 / 88236	1303	5534	Damenhose	200
310	A & B	Bahnhofstr. 2, 51465 Bergisch Gladbach			1560	6643	Sweatshirt	500
220	Grubersöhne	Feldgasse 17, 50899 Köln	Stefan Pux	0221 / 900200	1581	9821	Pullover	100
793	Henze und Partner	Mühlenweg 7, 51069 Köln	Michael Reitbach	0221 / 888881	1824	6643	Sweatshirt	500

Datensatz: 5 von 5

Bild: Tabelle Kunde mit allen Kunden- und Bestelldaten

Aus den genannten Gründen ist für eine professionelle Datenhaltung die hier gezeigte Tabelle Kunde nicht sehr sinnvoll. Die Probleme bei der Datenbearbeitung dieser Tabelle zeigen sich insbesondere, wenn eine Bestellung gelöscht oder eine Artikelbezeichnung geändert werden soll.
Falls Sie eine Bestellung löschen, dann löschen Sie gleichzeitig die dazugehörigen Firmendaten. Hat ein Kunde keine weiteren Bestellungen getätigt, so gehen die Firmendaten endgültig verloren. Und ändert sich der Artikelname (z.B. Damenschuhe in Damenstiefel), so müssen manuell alle Datensätze geändert werden, in denen dieser Artikel vorkommt.

Ziel eines optimalen Datenbankentwurfs ist es natürlich, die Tabellen so zu strukturieren, dass sich einerseits Informationen (in der Regel) nicht wiederholen, trotzdem aber alle relevanten Informationen aus den Tabellen gelesen werden können.

Durch die Normalisierung (vgl. Abschnitt 7.2.2. auf S. 214 ff) soll erreicht werden, dass keine Datenredundanz auftritt und trotzdem alle Informationen aus den Tabellen abrufbar sind. Um dieses Ziel zu erreichen, wird die Normalisierung stufenweise durchgeführt.

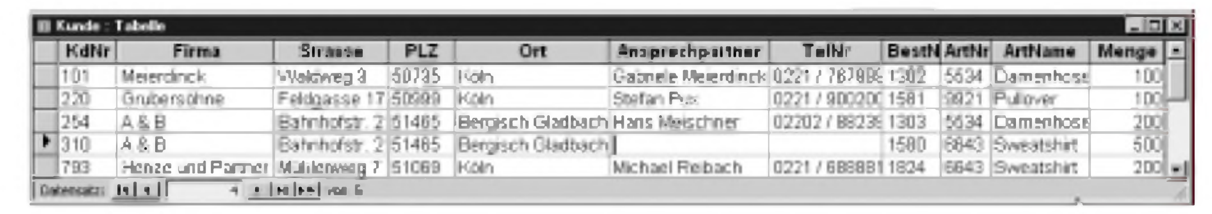
Wir erinnern uns (?): eine Tabelle befindet sich nur dann in der 1. Normalform (1NF), wenn in jedem Feld nur eine einzige, elementare Information steht.

In unserem Beispiel ist die Tabelle Kunde nicht in der 1. Normalform, da das Feld Adresse mehrere Informationen aufweist, nämlich die Informationen Straße, Postleitzahl und Ort.

Wir bringen diese Tabelle in die 1. Normalform, indem wir das Feld Adresse in die Felder Straße, PLZ und Ort aufteilen. Nach dieser Normalisierung ist jedem Feld nur jeweils eine Information zugeordnet.

---

[119] Wie bereits erwähnt: dies könnten Sie bei einem nicht zu großen Datenumfang auch noch in einer Tabellenkalkulationssoftware wie Excel oder Lotus 1-2-3 realisieren.



KdNr	Firma	Strasse	PLZ	Ort	Ansprechpartner	TelNr	BestN	ArtNr	ArtName	Menge
101	Meierdinck	Waldweg 3	50735	Köln	Gabriele Meierdinck	0221 / 78788	1302	5634	Damenhose	100
220	Grubersöhne	Feldgasse 17	50999	Köln	Stefan P[illegible]	0221 / 90020	1581	9921	Pullover	100
254	A & B	Bahnhofstr. 2	51465	Bergisch Gladbach	Hans Meischner	02202 / 8823	1303	5634	Damenhose	200
310	A & B	Bahnhofstr. 2	51465	Bergisch Gladbach			1580	6843	Sweatshirt	500
793	Henze und Partner	Mühlenweg 7	51069	Köln	Michael Reibach	0221 / 688881	1824	6643	Sweatshirt	200

Bild: Tabelle Kunde in 1.Normalform

Aber: Die Tabelle (in 1NF) enthält Redundanzen. Bestellt ein Kunde mehrere Artikel, so müssen jedesmal alle Kundendaten gespeichert werden!

Die Tabelle in der 1. Normalform enthält zwar in jedem Feld nur noch eine einzige, elementare Information. Aber es können sog. Anomalien oder Inkonsistenzen auftreten. Ändern Sie beispielsweise im ersten Datensatz für den Artikelnamen Damenhose die dazugehörige Artikelnummer und vergessen diese Änderung im fünften Datensatz, in dem dieser Artikel mit der dazugehörigen Artikelnummer ebenfalls zu finden ist, so treten Unstimmigkeiten auf, da dem Artikelnamen Damenhose unterschiedliche Artikelnummern zugeordnet sind! Noch eine Inkonsistenz: bei dem Kunden „A & B" ist einmal ein Ansprechpartner eingetragen, ein zweites Mal nicht.

Dieses Problem tritt deswegen auf, weil die Tabelle gleichzeitig verschiedenartige Informationen (Kundendaten, Bestelldaten, Artikeldaten) enthält, diese aber unabhängig voneinander geändert werden können.

Eine Relation befindet sich, wie bereits definiert, dann in zweiter Normalform (2NF), wenn jedes nicht zum Schlüssel gehörige Attribut funktional abhängig ist vom Gesamtschlüssel, nicht aber von Schlüsselteilen.

Der Schlüssel bei der hier gezeigten Tabelle Kunde ist dreiteilig und besteht aus KdNr, BestNr und ArtNr. (Wieso?) Aber: die Attribute Firma, Strasse, PLZ, Ort, Ansprechpartner und TelNr hängen nur von dem Teilschlüssel KdNr ab.

Wir müssen also für die 2.Normalform die Tabelle Kunde zerlegen: in eine neue Tabelle Kunde, in der nur von KdNr abhängige Attribute gespeichert werden, eine Tabelle Artikel (mit dem Artikelnamen) sowie eine Tabelle Bestellung, in der zu einer Bestellung gehörende Angaben verwaltet werden.

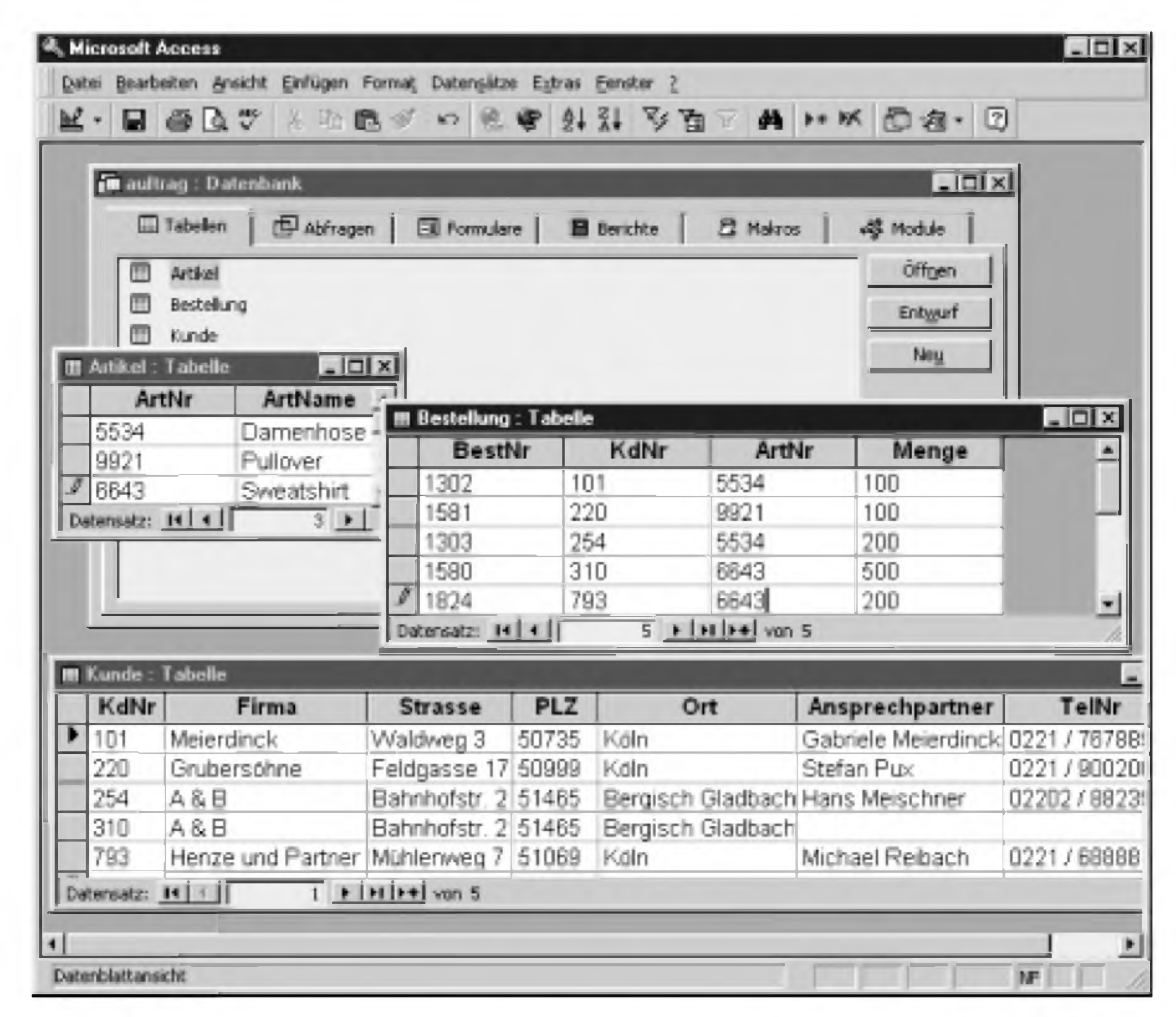
Kunde(KdNr, Firma, Strasse, PLZ, Ort, Ansprechpartner, TelNr)[120]

Artikel(ArtNr, ArtName)

Bestellung(BestNr, KdNr, ArtNr, Menge)

Die Attribute KdNr und ArtNr werden hier als Fremdschlüssel bezeichnet, denn es handelt sich dabei inhaltlich um Schlüsselattribute in anderen Tabellen.

[120] Die Schlüsselfelder sind in dieser Darstellungsform unterstrichen markiert.



ArtNr	ArtName
5534	Damenhose
9921	Pullover
6643	Sweatshirt

BestNr	KdNr	ArtNr	Menge
1302	101	5534	100
1581	220	9921	100
1303	254	5534	200
1580	310	6643	500
1824	793	6643	200

KdNr	Firma	Strasse	PLZ	Ort	Ansprechpartner	TelNr
101	Meierdinck	Waldweg 3	50735	Köln	Gabriele Meierdinck	0221 / 78788
220	Grubersöhne	Feldgasse 17	50999	Köln	Stefan Pux	0221 / 90020
254	A & B	Bahnhofstr. 2	51465	Bergisch Gladbach	Hans Meischner	02202 / 8823
310	A & B	Bahnhofstr. 2	51465	Bergisch Gladbach		
793	Henze und Partner	Mühlenweg 7	51069	Köln	Michael Reibach	0221 / 68888

Bild: Die drei Tabellen Kunde, Artikel und Bestellung (2.Normalform)

Wenn wir in die Tabelle Bestellung noch zwei weitere Attribute aufnehmen, Zahlungsart (z.B. „bar", „Scheck", „Überweisung") und Zahlungskennzeichen (als Kurzform für die Zahlungsweise, dann haben wir die im folgenden Bild gezeigte Situation.

Bestellung : Tabelle

BestNr	KdNr	ArtNr	Menge	Zahlungsken	Zahlungsart
1302	101	5534	100	1	bar
1303	254	5534	200	2	Scheck
1580	310	6643	500	3	Überweisung
1581	220	9921	100	3	Überweisung
1824	793	6643	200	2	Scheck

Datensatz: 5 von 5

Bild: leicht modifizierte Tabelle Bestellung

In diesem Fall müssen wir feststellen, dass das Zahlungskennzeichen bereits ausreicht, um über die Zahlungsart Auskunft zu geben. Anders formuliert: in der hier gezeigten Form kann Inkonsistenz dadurch auftreten, dass zu einem Zahlungskennzeichen ein inhaltlich falscher Wert für die Zahlungsart eingetragen werden kann. Gleichwohl ist die Tabelle in

2.Normalform, denn der Schlüssel ist elementar (und damit die Bedingung der 2NF automatisch erfüllt).

Diese Tabelle erfüllt jedoch nicht die 3.Normalform, denn es gibt ein Attribut (Zahlungsart), das bereits von dem Nicht-Schlüssel-Attribut Zahlungskennzeichen funktional abhängig ist.

Wieder muss also eine neue Tabelle ausgelagert und die Tabelle Bestellung verändert werden wie folgt.

Bestellung(<u>BestNr</u>, KdNr, ArtNr, Menge, Zahlungskennzeichen)

Zahlungsart(<u>Zahlungskennzeichen</u>,Zahlungsart)

Mit den so erhaltenen vier Tabellen haben wir das Problem normalisiert, die Tabellen liegen in 3.Normalform vor.

Anmerkung: Beachten Sie bitte, dass gleichwohl das Phänomen, dass ein und derselbe Kunde mit verschiedenen Kundennummern in den Datenbestand aufgenommen wurde, mit den hier vorgestellten Mitteln nicht beseitigt werden kann! Hierbei handelt es sich nämlich nicht um einen Widerspruch innerhalb des Datenbanksystems sondern um Semantik, also den vom Benutzer inhaltlich definierten Wunsch, dass ein bereits eingetragener Kunde nicht ein zweites Mal separat aufgenommen werden soll. Dies kann zum Beispiel mit speziellen Eingaberoutinen, die man innerhalb des Datenbanksystems definieren kann, überprüft werden.

8. INTERNET

Zum Thema *Internet* gibt es eine große Zahl von Dokumenten im Internet selbst, die dieses erläutern. Einige Erläuterungen werden im Folgenden gegeben, diese sollten jedoch durch tagesaktuelle Informationen direkt aus dem Internet ergänzt werden.

Daneben sei auf ein kleines Büchlein von [Tolksdorf] verwiesen, das grundlegende Informationen zum Aufbau und den Diensten des Internet liefert.

8.1. Historie

Das Internet hat seinen Ursprung im US-amerikanischen militärischen Sektor gegen Ende der sechziger Jahre. 1957 wurde die ARPA (Advanced Research Project Agency) gegründet zu einer Zeit, in der ein starkes Gefühl der Bedrohung durch die Sowjetunion bestand. Es war die Zeit des Kalten Krieges.

Das so entstandene ARPAnet wird als „Mutter des Internet" angesehen. Zwischen 1964 und 1972 wurde die paketvermittelte Übertragung entwickelt, d.h. Informationen werden in einzelne Päckchen aufgeteilt, die je nach Übertragungsprotokoll auch auf physikalisch vollkommen unterschiedlichen Wegen den Empfänger erreichen können. Für den Fall, dass ein Teil der Leitungen der Computer untereinander zerstört wären, könnten die Pakete doch noch den Empfänger erreichen. Dies war sehr wichtig für die militärische Kommunikation im Kriegsfall.

1969 wurden die ersten vier Knotenrechner des ARPAnet vom Department of Defense (DoD) zusammengeschaltet. Um eine Kommunikation auch noch bei einer teilweisen Zerstörung des Kommunikationsnetzes zu gewährleisten, wurde, anders als beim Telefon, keine Sterntopologie, sondern Ringvernetzung (Maschen) bevorzugt.

Ein weiterer Grund, dieses Netz aufzubauen, war die verstärkte Nachfrage der Wissenschaftler nach damals extrem teuren Computerresourcen. Es sollte die Möglichkeit geschaffen werden, Forscher mit entfernten Computerzentren zu verbinden, die ihnen einen Zugriff auf die Hardware- und Softwareresourcen erlaubte. Wenn der Plattenplatz, die Daten und auch die Rechenleistung der Computer gemeinsam benutzbar wurden, konnten diese Resourcen besser ausgenutzt werden.

Bis 1972 wurden auch die ersten Entwicklungen der heute Standard gewordenen Protokollfamilie TCP/IP (Transmission Control Protocol / Internet Protocol) spezifiziert und programmiert.

Zwischen 1974 und 1982 entwickelten sich parallel hierzu auch in Europa diverse Netze, z.B. die Netzwerke des Hahn-Meitner-Institutes Berlin (HMI-Net I und II) oder das BerNet; diese mündeten in das Deutsche Forschungsnetz DFN.

Ende der siebziger Jahre entstanden dann das Unix-to-Unix-Communications-Protocol (UUCP) und das Usenet.

1981 begannt das BITNet (Because it's time-Network) als kleines Netzwerk von IBM-Computern an der City University of New York (CUNY), das sich sehr schnell über die ganze Welt ausbreitete.

Nachdem die Kommunikation innerhalb eines Netzwerkverbundes im wesentlichen gelöst war, ging es anschließend um die Vernetzung der Netze.

1982 schlossen sich vier experimentelle Netze zum sog. Internetting Project zusammen, das bereits Internet genannt wurde. Diese waren ein paketvermitteltes Satellitennetz, ein paketvermitteltes Funknetz, das ARPAnet und ein Ethernet beim XEROX Research Center in Palo Alto.

Ebenfalls 1982 wurde die erste Distribution des 4.2 BSD Unix (Berkeley Systems Distribution) mit integriertem TCP/IP kostenlos von der University of Berkeley zur Verfügung gestellt. Auch deshalb kam es zu einer sehr schnellen Verbreitung an den Universitäten und unter den Studenten.

1983 spaltete sich das militärische MilNet vom ursprünglichen ARPAnet ab. Alle Knotenrechner sollten nun TCP/IP benutzen. Die Kontrolle ging an die DCA (Defense Communication Agency), heute bekannt unter DISA (Defense Information Systems Agency). Die beiden Netze blieben jedoch untereinander verbunden, und diese Verbindungen bekamen den Namen the DARPA Internet, was später zum Namen Internet verkürzt wurde. Über Satelliten waren schon damals Verbindungen ins europäische Ausland geschaffen worden. Hier spielte Skandinavien eine besondere Rolle. Durch das Nordunet wurde die Anbindung von Finnland aus über ganz Skandinavien verteilt. Auch heute noch ist Skandinavien (und insbesondere Finnland) ein besonders aktiver Teil des Internet.

Mitte der 80er begann auch die amerikanische National Science Foundation (NSF) Interesse am Internet zu zeigen. Um den Wissenschaftlern aller amerikanischen Universitäten den Zugang zum Netz zu ermöglichen, gründete sie das NSFNET. Um immer mehr Institutionen anzuschließen und einem NSFNET immer weiter zunehmenden Verkehr gerecht zu werden, wurde ein System, basierend auf Backbones (=Rückgrat) realisiert, das die großen Rechenzentren miteinander verband. An diese konnten sich andere eigenständige Campus- und Weitverkehrsnetze (WAN, Wide Areas Networks) anschließen. Dieser Backbone trägt heute mit seinen 45 Mbit/s-Anschlüssen die Hauptlast des Internetverkehrs. Damit übernahm die NSF immer mehr die Aufgaben des Arpanet, das schließlich Ende 1989 vom Department of Defense aufgelöst wurde.

1986 entstand der DNS-Namensraum (DNS = Domain Name System), der eine für damalige Verhältnisse sehr große Zahl von Rechnern adressieren konnte (bzw. kann).

1993 entstand beim CERN (Centre Européen de Recherche Nucléaire) in Genf das World Wide Web (WWW), das heute aufgrund seiner bunten Bildchen auch Neulingen mit entsprechenden Browser-Programmen offensteht und für viele als „das Internet" schlechthin gilt.

8.2. Dienste des Internet

Wir wollen zur Entspannung, bevor wir mit der Aufzählung der wichtigsten Dienste des Internet fortfahren, mit einigen, zum Teil schon etwas älteren, Presseschnipseln beginnen, die den Facettenreichtum des Internets belegen sollen.

- Seattle (dpa, Juni 1997) - Ein kalifornischer Jungunternehmer, dessen Firma im Internet unter der Adresse www.microsoftnetwork.com residiert, soll seine Adresse ändern. Das

verlangt der Softwarehersteller Microsoft aus Redmond bei Seattle (US-Bundesstaat Washington), dessen Sprecher am Dienstag betonte, es liege "ein bewusster Versuch zur Taeuschung von Internet-Nutzern vor."
Der kalifornische Unternehmer hatte den umstrittenen Namen zusammen mit anderen, die an Microsoft-Produkte erinnern, bei einem Händler erworben, der werbeträchtige Online-Adressen verkauft.
Hinter der von Microsoft kritisierten Adresse steckt eine von Bill Gates' Unternehmen unabhängige Firma, die die Herstellung und Betreuung von Webseiten anbietet. Der Besitzer der "Global Net Web Site Construction Company" ist ein kalifornischer Student.

- Düsseldorf (dpa, Juni 1997) - Die Deutschen hinken nach Angaben des Bundesverbandes des Groß- und Aussenhandels (BGA) bei der Nutzung moderner Kommunikationstechniken hinterher.
  Nur drei von 1 000 Deutschen hätten beispielsweise einen Internet-Anschluss. In den USA seien es dagegen mit zwölf Einwohnern viermal so viele. Selbst das kleine Island sei Deutschland mit 18 Internetanschlüssen je 1 000 Einwohner um Längen voraus. Diese Zahlen nannte BGA-Präsident Michael Fuchs bei einem Kongreß in Düsseldorf.
  Fuchs machte die Abschottung des deutschen Kommunikationsmarktes und überhöhte Preise dafür verantwortlich. In den USA zahlten Bürger für einen pauschalen Internet-Zugang 20 Dollar. Dafür könne ein deutscher Internetnutzer gerade einmal drei Stunden "surfen".
  BGA forderte, den deutschen Telekommunikationsmarkt konsequent zu liberalisieren und mehr Wettbewerb zu ermöglichen. Auch die Tarifparteien müssten sich auf die neue Arbeitsformen in den Zukunftsbranchen einstellen. Fuchs bekräftigte seine Forderung, Löhne und Gehälter nach unten zu öffnen, um so Jobs für weniger qualifizierte Dienstleistungen zu schaffen. "Es ist doch besser, überhaupt einen Job zu haben als gar keinen. Uns geht in Deutschland die Arbeit nicht aus. Sie ist einfach nur zu teuer." Auch müssten die Unternehmen die Arbeitszeit wählen können, die sie bräuchten.

- Frankfurt/Main (dpa, Juni 1997) - Mehrere Tausend Reiselustige haben sich in einem "virtuellen Auktionsraum" an der ersten Flugticket-Auktion der Lufthansa im Internet beteiligt. Rund 7000 Interessenten hatten sich für die Aktion registriert, bei der die Lufthansa an zwei Tagen insgesamt 66 Flugscheine unter den elektronischen Hammer bringen will.
  Die Fluggesellschaft möchte mit diesem Marketing-Instrument ihren "Infoflyway" mit der Möglichkeit zum Buchen im Internet bekannter machen. Längerfristig sollen für die Airline dabei kräftige Einsparungen herausspringen. Bisher ist die Zahl der Online-Buchungen noch relativ gering: Zwischen dem Start im November 1996 und März 1997 wurden nach Angaben der Airline rund 10 000 Flugscheine auf diese Weise geordert. Lufthansa(http://www.lufthansa.com) ist damit jedoch nach eigenen Angaben in Europa Marktführer, wobei International derzeit nur "eine Handvoll" Airlines diesen Service anböten.
  "Überwältigend" war nach Angaben der Lufthansa-Sprecherin Dagmar Rotter der Andrang bei der Ticket-Versteigerung im Internet. Die Teilnehmer konnten sich für ein Mindestgebot von zehn DM am Computer per Mausklick an der Ersteigerung beteiligen. Die Liste der Ziele enthielt exotische Orte wie Bangkok, Johannesburg, Kuala Lumpur oder Singapur genauso wie die näherliegenden Amsterdam, Paris oder Rom. Nach Angaben Rotters konnten die ersten Gewinner "echte Schnäppchen" machen, die zum

Teil weniger als die Hälfte des günstigsten derzeit am Markt gehandelten Ticketpreises betrugen.
Eine höhere Zahl von Online-Buchungen soll Lufthansa Einsparungen unter anderem im Marketing, bei den Reisebüro-Provisionen und im Service-Bereich bringen. Über die Höhe schweigt sich die Airline - mit dem Hinweis auf das Interesse der Konkurrenz - allerdings aus. Genutzt werde das Medium vermutlich vorwiegend von Geschäftsleuten. Business-Class-Tickets machten rund 60 Prozent der so verkauften Flugscheine aus.

- Berlin (dpa, Juni 1997) - Die positive Wirtschaftsentwicklung der Hard- und Softwarebranche darf nach Ansicht der Hersteller nicht durch das geplante Gesetz zur Verhinderung von Datenverschlüsselung beeinträchtigt werden. Ein solches Gesetz sei der kostspielige Versuch, "zu kontrollieren, was faktisch nicht zu kontrollieren ist", sagte der Vorsitzende des Fachverbandes Informationstechnik im VDMA und ZVEI, Jörg Menno Harms, am Donnerstag in Berlin. Bundesinnenminister Manfred Kanther hatte Ende April angeregt, für elektronische Netze wie das Internet eine Überwachung zu ermöglichen.
  Die Nutzung von Verschlüsselungen, sogenannter Kryptographie, in Datennetzen lasse sich zwar einschränken, der kriminelle Mißbrauch allerdings nicht, meint der Fachverband. Den Preis einer Regulierung hätten die deutsche Wirtschaft und die Privatanwender zu zahlen. Denn sie müssten eine aufwendige Infrastruktur zur Schlüsselarchivierung finanzieren. Stellen, bei denen Entschlüsselungs-Code hinterlegt werden, seien lohnende Angriffsziele zum Beispiel von ausländischen Nachrichtendiensten.
  Die organisierte Kriminalität werde vorhandene Umgehungsmöglichkeiten nutzen, heißt es. Denn im Gegensatz zum Telefonverkehr könne im Internet mit zwei Mausklicks ein wirksamer aber unbemerkter Schutz gegen Überwachungsmaßnahmen installiert werden. Geheime Botschaften seien in Bild-, Film- oder Musikdateien sowie im elektronischen Rauschen praktisch unauffindbar zu verstecken.
  Die Branche befürchtet zudem, dass der lukrative Zukunftsmarkt des elektronischen Handels durch falsche Rahmenbedingungen beeinträchtigt werden könnte. Ohne Beeinträchtigungen sei mit einem Marktwachstum von derzeit 1,6 Milliarden DM auf 14 Milliarden DM im Jahre 2000 zu rechnen.
  Die Informationstechnik in Deutschland verzeichnet weiterhin gute Zuwachsraten. Der heimische Markt für Hard- und Software und Dienstleistungen stieg 1996 um fünf Prozent auf 80,5 Milliarden DM. Für das laufende Jahr werde aufgrund einer steigenden Nachfrage im Service-Bereich mit einem Plus von sechs Prozent auf 85,6 Milliarden DM und 1998 mit einer siebenprozentigen Steigerung auf 92 Milliarden DM gerechnet. Der Wegfall europäischer Importzölle am 1. Juli 1997 fuer Deutschland im Wert von 650 Millionen DM kann laut Fachverband bei PCs und Druckern langfristig zu Preissenkungen zwischen drei und fünf Prozent führen.

- (com! Newsflash, Januar 1998) T-Online hat das Jahr 1997 mit einem Rekordwachstum von ueber 550.000 neuen Teilnehmern abgeschlossen. Damit hat der Online-Dienst der Deutschen Telekom jetzt ueber 1,9 Millionen Kunden und ist nach eigenen Angaben der mitglieder- und wachstumsstaerkste Online-Service in Deutschland. Allein im zweiten Halbjahr 1997 wurden ueber 350.000 Neukunden gewonnen. Das durchschnittliche Monatswachstum lag 1997 bei 45.000 neuen Teilnehmern - sicher auch ein Verdienst der agressiven und flaechendeckenden Werbekampagne. Im Dezember 1997 erreichte die

Zahl der Verbindungen mit ueber 52 Millionen einen neuen Hoechstwert. Die Gesamtzahl der Verbindungen lag 1997 sogar bei rund einer halben Milliarde. Gegenueber dem Vorjahr bedeutet dies ein Wachstum von 60 Prozent. Die Zahl der Online-Konten hat sich fast verdoppelt: Waren es 1996 noch 1,8 Millionen, so werden aktuell bereits rund 3,5 Millionen elektronische Bankkonten gefuehrt.

- (Auszug einer Greenpeace-Presseerklärung, 29.01.1998) Landentsorgung der "Brent Spar" ist Praezedenzfall
  By email: From: redaktion@greenpeace.de - Date: Thu, 29 Jan 1998 12:06:36 +0100 (MET)
  Proteste haben ihr Ziel erreicht - Greenpeace fordert jetzt generelles Versenkungsverbot
  Hamburg, 29.1.98: "Die Vernunft hat gesiegt", sagt Birgit Radow, Geschaeftsfuehrerin von Greenpeace Deutschland zur heutigen Entscheidung von Shell, die "Brent Spar" an Land zu entsorgen.
  "Auf diese Shell-Entscheidung haben wir seit zweieinhalb Jahren gewartet. Der Beschluss von Shell bestaetigt, dass die durch Millionen Menschen unterstuetzten Greenpeace- Aktionen gegen die Versenkung der "Brent Spar" richtig waren. Die Entscheidung ist ein Erfolg fuer den Schutz der Meere und ein erster Schritt hin zu einem generellen Versenkungsverbot von Oelplattformen."
  Shells heutige Entscheidung ist ein Praezedenzfall, bestaetigt Greenpeace. "Das setzt das richtige Signal, denn ueber vierhundert Plattformen in Nordsee und Nordostatlantik werden in den naechsten Jahrzehnten ebenfalls entsorgt werden muessen",sagt Greenpeace-Oelexperte Christian Bussau. "Jetzt sind Shell, Esso, BP und die anderen Oelfirmen gefordert, detaillierte Plaene fuer die Landentsorgung dieser vielen Plattformen auf den Tisch zu legen." (...) [121]
- (telepolis, Januar 1998) Internet-Sucht und die Studenten (von Florian Roetzer 26.01.98):
  Internet hält die Studenten vom Arbeiten ab.
  Eine Untersuchung aus England, veröffentlicht in einer neuen Zeitschrift, die sich der Psychologie und dem Verhalten im Cyberspace-Zeitalter widmet, warnt davor, dass das Internet auf das Studium einen schlechten Einfluß haben könnte.
  (Der ganze Artikel kann bei Telepolis unter der Adresse http://www.heise.de/tp/deutsch/inhalt/glosse/2254/1.html angesehen werden.)

Sehen wir uns im Überblick einmal die wichtigsten Dienste an. Ähnliche und umfangreichere Übersichten gibt es im Internet zahlreiche; exemplarisch seien hier die Links bei DINO online [DINO] erwähnt.

8.2.1. Elektronische Post (electronic mail)

Electronic Mail (elektronische Post) stellt einen der wohl wichtigsten und am meisten benutzten Netzwerkdienste dar, nämlich die Möglichkeit, auch über große Entfernungen und verschieden Zeitzonen hinweg relativ schnell und sicher zu kommunizieren. Analog zur Briefpost werden hierbei Empfänger und Absender einer Nachricht über eine Adresse identifiziert. Diese setzt sich wie folgt zusammen: `nutzername@domain-name` Hierbei

[121] Da bei elektronischer Mail beim Wechsel zwischen den verschiedenen Rechnersystemen Umlaute nicht immer sauber oder eben gar nicht konvertiert werden, schreiben viele „Insider" (so wie hier die Greenpeace-Aktivisten in ihrer Mail) statt „ä" „ae" usw.

wird der Nutzername in vielen Mehrbenutzersystem mit dem Login-Namen gleichgesetzt, während der Domain-Name das Internet-Interface der entsprechenden Maschine, auf der der Empfänger die Nutzerkennung innehat, spezifiziert.

Wie schon erwähnt, können Gateways zu anderen Netzen außerhalb des Internet angesprochen werden, so dass der Dienst E-Mail bei weitem nicht nur ein Internetdienst ist. Es existieren zwei Möglichkeiten, um Post in Netze außerhalb des Internet zu schicken. Zum einen können viele Gateways zu fremden Netzen als High-Level Domain angesprochen werden. Entsprechend wird eine Mail in folgendem Format adressiert:
`nutzername@fremdnetz-maschinenname.fremdnetz`

In einer E-Mail werden in der Regel textuelle Nachrichten an den Adressaten geschickt; dieser kann nach dem Einloggen auf seinem Rechner mit Hilfe eines Clientprogrammes inzwischen eingetroffene Post durchlesen, ablegen, löschen oder in seinem Dateisystem abspeichern. Erreicht eine Mail ihren Adressaten nicht, so wird sie mit einem entsprechenden Vermerk an den Absender zurückgeschickt. Zusätzlich zu der textuellen Nachricht kann einer Mail eine beliebige Datei als „Enclosure” („Attachment”, „Anlage”) beigefügt werden, wobei binäre Formate wie beispielsweise Programme meist konvertiert werden, um kritische, normalerweise nicht übertragbare Zeichen durch unproblematische Zeichen zu ersetzen. Entsprechend konvertierte Dateien müssen auf der Zielmaschine rückübersetzt werden.

8.2.2. Mailing-Listen

Soll eine Mail mehrere Teilnehmer erreichen, zum Beispiel die Mitglieder einer Diskussionsrunde, so besteht die Möglichkeit, als Ziel alternativ eine Liste mit den Adressen aller Teilnehmer zu verwenden bzw. die Mail an den Verwalter einer solchen Liste zu schicken. Entsprechend wird die Mail an alle aufgeführten Adressen weitergeleitet. Trifft der Listenverwalter dabei eine Auswahl unter deneingegangenen Beiträgen, so spricht man von einer moderierten Liste. Um an einer Diskussion innerhalb einer solchen Gruppe teilzunehmen, ist es nur notwendig, sich in die entsprechende Verteilerliste eintragen zu lassen. Mailing-Listen können außerdem zur Verbreitung von elektronischen Journalen im Netz verwendet werden. Mailing-Listen können manuell geführt und verwaltet werden; eine automatische Verwaltung realisiert *Listserv*. Der einzelne Nutzer kann sich über bereitgestellte Kommandos selbst aus der Liste aus- oder sich in diese eintragen. Nebenher archiviert Listserv eingegangene Diskussionsbeiträge durch sogenannte Log-Files und ermöglicht es so, auch später in Diskussionsrunden einzusteigen bzw. sich über deren Verlauf zu informieren.

8.2.3. News (Usenet Diskussionsgruppen)

NetNews und Newsgroups setzen im Gegensatz zu Listserv nicht voraus, dass man sich in eine Diskussionliste einträgt, um entsprechende Beiträge zu erhalten. Sie wenden sich eher an einen Nutzer, der zu einem bestimmten Zeitpunkt Informationen und Meinungen zu einem bestimmten Thema gesammelt erhalten möchte. Mit Hilfe eines entsprechenden Hilfsprogrammes, einem Newsreader, können die nach Themen hierarchisch geordneten Newsgroups nach interessanten Gebieten auswählt und die Beiträge bzw. Artikel gelesen und auf den eigenen Rechner kopiert werden. Die sogenannten *Newsgruppen* oder *Usenet News* bieten eine schwarzen Brettern nachempfundene Möglichkeit, Diskussionsbeiträge, Angebote und Kommentare weltweit und online auszutauschen.

8.2.4. Telnet - interaktives Arbeiten mit entfernten Rechnern

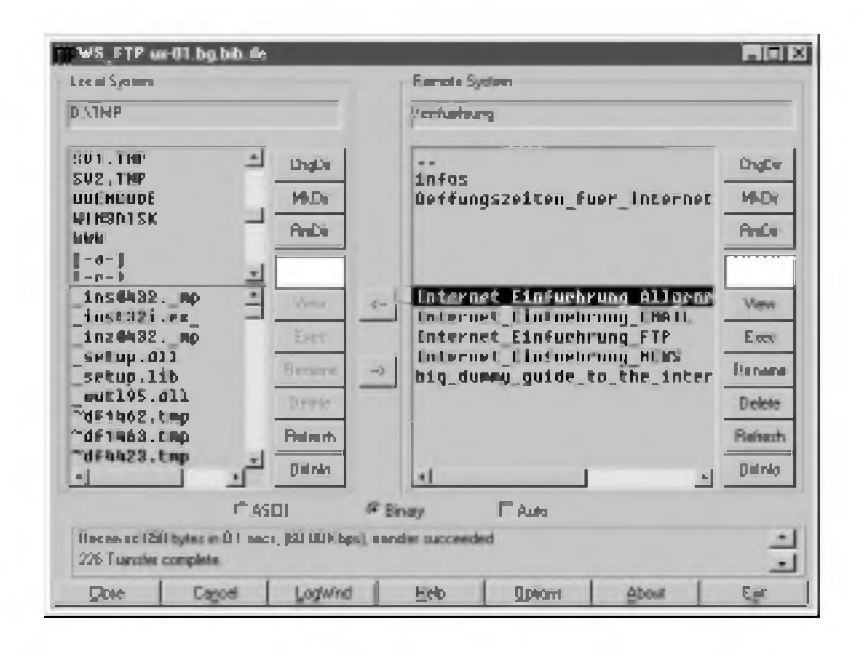
Der Service Telnet bildet einen weiteren wichtigen Basisdienst im Internet. Mittels einer Telnetverbindung kann über das Internet ein Remote-Login auf anderen Rechnern durchgeführt werden. Telnet wird für mehrere Aufgaben genutzt: es ist möglich, auch über große Entfernungen auf den eigenen Rechnerbereich zuzugreifen, um in der gewohnten Arbeitsumgebung zu sein und benötigte Dateien und Werkzeuge zu nutzen oder einfach nur die inzwischen angekommene Mail zu lesen. Ungenutzte Systemressourcen und CPU-Zeiten von Superrechnern können einem größeren Kreis von Nutzern zur Verfügung gestellt. Primär werden Telnet-Verbindungen jedoch dazu verwendet, um auf fremde Informationssysteme und Literaturdatenbanken über das Internet zuzugreifen.
Früher, bevor der PC seinen Zug durch die Welt angetreten hat, war es üblich, dass auf einen (Groß-)Rechner über ein sogenanntes Terminal zugegriffen wurde: ein Bildschirm mit einer Tastatur.
Daher leitet sich der Begriff Terminalemulation ab: ein Programm, das die Funktionalität eines Terminals bietet.
Der Netscape Navigator (und andere Webbrowser) ermöglichen nicht direkt eine Telnet-Verbindung, es können aber in der Regel externe Programme (wie das mit Windows für Workgroups oder Windows NT mitgelieferte `telnet.exe`) eingebunden werden.

8.2.5. FTP - File Transfer Protocol (Dateitransfer)

Via FTP kann ein Anbieter allgemein Dateien im Internet zur Verfügung stellen. Der an einer bestimmten Datei interessierte Nutzer loggt sich auf dem FTP-Server der Maschine ein, welche den Dienst und die Datei anbietet.

Viele FTP-Server bieten einen allgemeinen Zugang, das heißt, man loggt sich mit "anonymous"' als Benutzernamen und der eigenen E-Mail-Adresse als Password ein. In diesem Fall spricht man von anonymem FTP. Der FTP-Server bietet dem eingeloggten Nutzer Kommandos zum Sichten der zur Verfügung stehenden Dateien, die wie in einem Dateisystem angeordnet sind, durch das der Nutzer navigieren kann.
Interessante Dateien können auf den eigenen Rechner übertragen werden; zusätzlich kann der Nutzer oft eigene Dateien an den FTP-Server übertragen.
Bei der Datenübertragung stehen zwei Übertragungsmodi zur Verfügung: der ASCII-Mode dient zur Übertragung einfacher Textdateien, wobei der Zeichensatz auf verschiedenen

Betriebssystemen entsprechend auf der Zielmaschine umgewandelt werden. Der Binary-Mode lässt die Datei unverändert und wird zum Beispiel für die Übertragung von Programmen oder Grafikformaten verwendet.



Das File Transfer Protocol (ftp) dient zum Austausch von Dateien zwischen verschiedenen Rechnern. Dieser Dienst kann mit eigenen Programmen wie beispielsweise WS_FTP unter Windows (siehe Bild oben) verwendet werden; ftp kann aber auch innerhalb der WWW-Browser (Netscape Navigator, Opera etc.) genutzt werden.

Wo meist das anzugebende Übertragungsprotokoll `http://` lautet, wird in solch einem Fall `ftp://` angegeben.

8.2.6. Archie (Suchen auf ftp-Servern)

Um auf Dateien via FTP zugreifen zu können, müssen diese erst lokalisiert werden, das heißt, der in Frage kommende FTP-Server und der Pfad der Datei müssen spezifiziert werden. Archie ist ein einfaches Hilfsprogramm, welche es erlaubt, Suchanfragen nach bestimmten Dateinamen oder Teilen daraus zu stellen. Entsprechend liefert Archie Serveradressen und Pfade möglicher Dateien. Eine Suche über Archie setzt voraus, das der Dateiname genügend über den Inhalt der Datei aussagt. Sowohl FTP als auch Archie können auch via Mail genutzt werden. Bestimmte Server im Internet interpretieren den Inhalt eingehender Mail als FTP- bzw. Archie-Kommandos und antworten dem Absender durch eine Mail, mit der auch angeforderte Dateien übertragen werden.

8.2.7. Gopher (textorientiertes, hierarchisches Menüsystem)

Gopher ist ein hierarchisch angeordnetes Informationssystem, das aus Textdokumenten besteht, die keine Hyperlinks (das heißt keine mit der Maus oder Tastatur anwählbaren Querverbindungen) wie dieWWW-Dokumente enthalten.

Gopher ist zwar nicht so schön bunt wie das World Wide Web, aber weltweit gibt es doch (noch) eine große Anzahl von Gopher- Servern, auf denen - vor allem im wissenschaftlichen Bereich - umfangreiche Informationen bereitliegen.

8.2.8. WWW - World Wide Web

Das World Wide Web verknüpft - ähnlich wie Gopher, allerdings graphisch aufgepäppelt - verteilt im Internet vorliegende Quellen durch Hypertextverbindungen. Durch Grafiken, unterschiedliche Schriftformen oder Zahlen gekennzeichnete Begriffe können ausgewählt werden, um über die Hypertext-Referenz das nächste Dokument aufzurufen. Der Nutzer kann so durch den Informationsraum navigieren, wobei auch auf verschiedenen Rechnern abgelegte Daten unter einer einheitlichen Oberfläche zugänglich sind und assoziativ miteinander verknüpft werden.
Das World Wide Web entstand 1993 bei CERN (siehe hierzu [CERN] und [W3C] im Literaturverzeichnis).

8.2.9. IRC - Internet Relay Chat

Das Internet Relay Chat (IRC) beruht im Gegensatz zu den News oder ListServ-Listen nicht auf dem Austausch von abgelegten Nachrichten, sondern erlaubt eine direkte Kommunikation mit allen Online-Partnern über das Internet. IRC ist in verschiedene Kanäle aufgeteilt, die jeweils unter einem bestimmten Thema oder Szenario organisiert sind und zum Beispiel als Diskussionsforum oder auch zum einfachen Smalltalk benutzt werden.

Zum Thema IRC (Chat) gibt es unter http://irc.pages.de/ und auf der Seite http://virtual-village.de/internet/irc/ weitere Informationen. Außerdem gibt es eine Übersicht zum Stichwort *Chat* bei web.de.

8.3. Intranet

Unter einem *Intranet* versteht man die Nutzung der mit dem Internet aufgekommenen Kommunikations- und Informationsstrukturen, allen voran das Übertragungsprotokoll TCP/IP, auf das in den Online-Unterlagen[122] noch eingegangen werden wird.
Dabei werden die Daten, Informationen und Kommunikationsmöglichkeiten jedoch lediglich firmenintern - nicht unbedingt aber nur hausintern - zur Verfügung gestellt. Für weitere Informationen zum Thema Intranet sei auf die Informationen von Jürg Keller hingewiesen, die im World Wide Web[123] abgerufen werden können.

---

[122] Siehe: http://www.bg.bib.de/~fhdwbm/work/internet/proto.htm

[123] Die Adresse der Seiten von Jürg Keller lautet http://www.simsy.ch/intranet/.

8.4. Multimedia im Internet: Sound und Video

Vorhandene Sound und Videodateien können verhältnismäßig einfach auf einer WWW-Seite eingebunden werden. Zum Abspielen muss der entsprechende Webbrowser entweder das Dateiformat direkt unterstützen oder ein Programm, ein sogenanntes *Plug-In*, aufrufen. Es gibt mittlerweile viele Zusatzprogramme, mit denen fast jedes Dateiformat abgespielt werden kann. Die verschiedenen Formate haben unterschiedliche Vorzüge und auch Nachteile. Wenn Sie selber Webseiten entwickeln wollen, dann sollten Sie die meistverbreiteten Dateiformate kennen.

8.4.1. Einige Dateiformate für Video und Audio

Nachfolgend seien die gebräuchlichsten Dateiformate in den Bereichen Video und Audio kurz aufgeführt[124].

AIFF (Audio Interchange File Format)
Das AIFF ist das Standarformat für den Macintosh. Es wurde 1988 von Apple entwickelt und basiert auf dem EA IFF 85 Standard for Interchange Format File von Electronic Arts.

WAV
Ist das Standardformat für Windows-Rechner und basiert ebenfalls auf dem EA IFF 85 Standard.

AU
Dieses ist der Standard auf Unix-Rechnern. Aufgrund seiner recht guten Kompressionsrate wurde es zum Standard im Internet. Jeder Browser unterstützt dieses Format.

MPEG Audio (MPEG 1) (Motion Picture Expert Group)
Als ein neuer Standard scheint sich der MPEG-1 Layer 3 durchzusetzen. Durch pychoakustische Kompressionsalgorithmen können Kompressionsraten von 12:1 erreicht werden. Allerdings benötigt man zum Abspielen spezielle Software, deren Decoderalgorithmen sehr rechenintensiv sind.

MIDI
MIDI ist ein Standard für die Ansteuerung MIDI-fähiger Musikinstrumente. MIDI-Dateien enthalten Informationen wie Tonhöhe und -länge, sind also eine Art elektronisches Notenblatt. Mittlerweile ist fast jede Soundkarte mit einem midi-kompatiblen Soundchip ausgerüstet. MIDI-Dateien zeichnen sich durch ihre geringe Größe aus und werden daher oft als Hintergrundmusik für Webseiten genutzt.

Quicktime
Quicktime ist der Standard für Videodateien auf dem Macintosh. Es wurde von der International Standards Organisaton (ISO) als Dateiformat für den neuen Multimedia-Standard MPEG-4 bestimmt.

AVI (Video for Windows)
AVI-Dateien sind der Videostandard auf Windows-Rechnern.

[124] An dieser Stelle noch einmal herzlichen Dank an meinen Kollegen Dr. Stefan Nieland von der FHDW Paderborn; er hat diesen Abschnitt zusammengestellt.

MPEG Video (MPEG 2)
MPEG Video ist ein systemübergreifendes Dateiformat mit sehr hohen Kompressionsraten. Zum Abspielen benötigt man eine hohe Rechnerleistung oder eine geeignete MPEG-Steckkarte.

Streaming Media
Normalerweise muss eine Datei komplett aus dem Internet geladen werden, bevor sie betrachtet werden kann. Streaming Media schaft hier Abhilfe:

1. der Computer wird zum Radio bzw. zum Fernseher
2. durch immer schnellere Übertragungsraten und bessere Kompressionsalgorithmen gelingt es, Musik und Videos live auf dem Computer abzuspielen.

Produkte

1. RealAudio/Real Video von Progressive Networks
2. VivoActive von Vivo-Software inc.
3. LiquidAudio von Liquid Audio Inc.
4. VDOLive von VDOnet Corporation
5. Netshow von Microsoft Corporation

VRML
VRML ist das Akronym für *Virtual Realitiy Modelling Language*. Mit Hilfe dieser Sprache lassen sich dreidimensionale Räume und Gegenstände beschreiben. Es lassen sich dabei Lichtquellen, Kamerapositionen, Oberflächen, Bewegungen und auch Töne definieren.

Genau wie HTML liegen VRML-Dateien in einfachen Textdateien vor. Zum Betrachten bzw. zum Durchwandern der VRML-Welt braucht mach einen VRML-Browser. Einer der prominentesten VRML-Browser ist der Cosmo-Player von Silicon Graphics. Er wird vom normalen Internetbrowser aufgerufen, sobald man auf eine Seite mit VRML-Daten stößt.

8.4.2. Graphikformate im WWW

Für computererzeugte Graphiken wird im WWW hauptsächlich das GIF-Format verwendet (Graphics Interchange Format), das eine maximale Farbtiefe von 8 Bit oder 256 Farben gestattet. Merkmale des GIF-Formats:

1. Verwendung des Lempel-Ziv-Welch (LZW) Kompressionsalgorithmus, der eine verlustfreie Reduzierung des Datenvolumens der Graphik gewährleistet (vgl. Abschnitt 1.6. auf Seite 46).
2. GIF-Bilder werden umso besser komprimiert, je mehr gleichfarbige Sequenzen sie enthalten

LZW: Horizontale Sequenzen gleichfarbiger Pixel werden durch eine Zahl ersetzt, die die Länge der betreffenden Sequenz definiert, während identische horizontale Linien zusätzlich zeilenweise komprimiert werden. Das Kompressionsverhältnis liegt beim LZW-Verfahren überlicherweise bei 4:1.

Neben dem GIF- wird auch das JPG-Format im World Wide Web eingesetzt. Das JPG-Format heißt richtig JPEG-Format (Joint Photographic Expert Group). Das Komprimierungsverfahren basiert auf der Trennung von Farbtönen und der Helligkeit, wobei das gesamt Bild in Bereiche unterteilt wird. Für die Kompression wird dabei eine relativ exakte Schwarzweißkopie des Originals erstellt, während detaillierte Farbnuancen, die vom Auge (im Idealfall) ohnehin nicht wahrnehmbar sind, verworfen werden. Praktisch bedeutet das, das Bild wird mit zunehmender Kompression unschärfer.

Im Unterschied zum GIF-Format resultiert die JPEG-Kompression daher in jedem Fall in einen Qualitätsverlust. Insgesamt bewegt sich das Kompressionsverhalten dabei zwischen 10:1 und 100:1.

Außer JPEG und GIF wird auch das (nicht mehr so ganz) neue Format PNG (Portable Network Graphics) zunehmend eingesetzt, das die Vorteile von GIF und JPEG bündeln soll.

8.5. Einführung in HTML

Aus Platzgründen[125] wird an dieser Stelle kein HTML-Kurs wiedergegeben; einen solchen, vielleicht den besten, finden Sie als Spiegelung der Originalquellen[126] eines Kurses von Stefan Münz unter der URL http://www.bg.bib.de/FHDW/Intranet/selfhtml/selfhtml.htm bzw. unter http://www.pb.bib.de/FHDW/Intranet/selfhtml/selfhtml.htm.

[125] Die Online-Materialien zur Seitenbeschreibungssprache HTML im FHDW-Intranet finden Sie unter der Adresse http://www.bg.bib.de/FHDW/Intranet/winformatik/04inet3.htm.

[126] Die Original-URL der Seiten von Stefan Münz ist http://www.netzwelt.com/selfhtml/ .

A. Anhang

A.1. Auszug aus einer ASCII-Tabelle

Nachstehend wird ein Auszug aus der in Deutschland unter Windows üblichen ASCII-Tabelle (sogenannter ANSI[127]-Code) wiedergegeben.

32		33	!	34	"	35	#	36	$	37	%	38	&	39	'	40	(	41	)
42	*	43	+	44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=
62	>	63	?	64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O	80	P	81	Q
82	R	83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_	96	`	97	a	98	b	99	c	100	d	101	e
102	f	103	g	104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w	120	x	121	y
1z		1†124	|	†126	~††12¬	129	▯	130	,	131	ƒ								
132	„	133	…	134	†	135	‡	136	ˆ	137	‰	138	Š	139	‹	140	Œ	141	▯
1'Z		143	▯	144	▯	145	‘	146	’	147	“	148	”	149	•	150	–	151	—
1▯2	˜	153	™	154	š	155	›	156	œ	157	▯	158	z	159	Ÿ	160		161	¡
162	¢	163	£	164	¤	165	¥	166	¦	167	§	168	¨	169	©	170	ª	171	«
172	¬	173	-	174	®	175	¯	176	°	177	±	178	²	179	³	180	´	181	µ
182	¶	183	•	184	¸	185	¹	186	º	187	»	188	¼	189	½	190	¾	191	¿
192	À	193	Á	194	Â	195	Ã	196	Ä	197	Å	198	Æ	199	Ç	200	È	201	É
202	Ê	203	Ë	204	Ì	205	Í	206	Î	207	Ï	208	Ð	209	Ñ	210	Ò	211	Ó
212	Ô	213	Õ	214	Ö	215	×	216	Ø	217	Ù	218	Ú	219	Û	220	Ü	221	Ý
222	Þ	223	ß	224	à	225	á	226	â	227	ã	228	ä	229	å	230	æ	231	ç
232	è	233	é	234	ê	235	ë	236	ì	237	í	238	î	239	ï	240	ð	241	ñ
242	ò	243	ó	244	ô	245	õ	246	ö	247	÷	248	ø	249	ù	250	ú	251	û
252	ü	253	ý	254	þ	255	ÿ												

[127] ANSI - American National Standardisation Institute, Normungsinstitut in den USA

A.2. Dateinamenendungen und Dateiformate

Nachstehend eine selbstverständlich höchst unvollständige Auflistung von Dateiformaten und den in der Regel damit assoziierten Dateinamenendungen (-erweiterungen, Extensions).

Programm / Sparte	Dateinamenendung	Bemerkungen
Textverarbeitung		
Microsoft Word für Windows	.doc	Proprietäres Format[128]
Lotus Word Pro	.lwp	Proprietäres Format; zahlreiche Importfilter[129]
Corel WordPerfect	(keine spezifische Endung)	Proprietäres Format; zahlreiche Importfilter
Tabellenkalkulation		
Microsoft Excel	.xls	Proprietäres Format, einige Importfilter
Lotus 1-2-3	.wks, .wk4, .123 (je nach Version)	Proprietäres Format, einige Importfilter
Graphik		
Corel Draw	.cdr	Vektorgraphik[130]; proprietäres Format
Adobe Photoshop	.phs, .tif	.phs: Bitmapgraphik[131]; proprietäres Format .tif: Bitmapgraphik; offenes Format
Paintbrush, Paint	.pcx, .bmp	offen gelegte Bitmapgraphik (-formate)
Paint Shop Pro (Shareware)	.pcx, .bmp, .jpg, .tif, .gif u.v.a.m.	beherrscht sehr viele Bitmapformate[132]

[128] Ein *proprietäres* Format ist ein firmenspezifisches, nicht offengelegtes Format; die meisten Programme anderer Hersteller können daher ein solches Format nicht oder nur schlecht lesen bzw. interpretieren.

[129] Unter einem *Importfilter* versteht man ein (mitgeliefertes oder separat einzuspielendes) Zusatzprogramm, mit dem man ein fremdes Dateiformat in ein Programm einlesen kann.

[130] Unter Vektorgraphik versteht man das Vorgehen, bei dem jeweils ein gesamtes Graphikobjekt bearbeitet wird; beispielsweise ist ein Kreis in einem Vektorgraphikprogramm als Objekt zu behandeln und kann jederzeit gelöscht, anders gefärbt oder sonstwie manipuliert werden.

[131] Im Gegensatz zur Vektorgraphik wird bei einer Bitmapgraphik jeder einzelne Bildpunkt isoliert behandelt und gespeichert. Ist ein Kreis einmal gezeichnet, so ist er bei einer Bitmapgraphik einfach die Summe seiner konkreten Bildpunkte. Ein Vergrößern oder Verkleinern oder Umfärben eines solchen Kreises erfordert einen sehr viel höheren Aufwand als bei einer Vektorgraphik.

[132] GIF und JPEG sind im Internet nutzbare Graphikformate.

Datenaustauschformate		
praktische alle Textprogramme	.txt	pures ASCII- oder ANSI[133]-Textformat, das in bezug auf die Formatierung nicht mehr als Zeilenumbrüche konservieren kann
zahlreiche Textverarbeitungs-programme	.rtf	RTF: Rich Text Format; von Microsoft vorgegebenes, offengelegtes Dateiformat zum Austausch von Textdokumenten
Internet-Browser u.a.	.htm, .html	HyperText Markup Language Seitenbeschreibungssprache für das Internet
Adobe Acrobat, Adobe Acrobat Reader	.pdf	Offengelegtes Format von Adobe[134], das als Datenaustauschformat durch den kostenfrei abgegebenen Acrobat Reader allerdings vor allem im Offline-Publishing (CD-ROM) weite Verbreitung gefunden hat.

[133] ASCII steht üblicherweise für den unter DOS benutzten Code; unter Windows wird ein ab Zeichen Nr. 128 anderer ASCII-Code eingesetzt, der dann in der Regel ANSI-Code genannt wird (ANSI=American National Standards Institute).

[134] Siehe hierzu die Erläuterungen auf den Webseiten von Adobe Deutschland: http://www.adobe.de/products/acrobat/adobepdf.html

A.3 PGP - Pretty Good Privacy

Nachstehend wird eine Quelle aus dem World Wide Web auszugsweise wiedergegeben. Die Original-URL ist http://www.tu-chemnitz.de/~hot/pgp.html. Autor der Seite ist Holger Trapp. Der Text basiert auf der letzten dokumentierten Änderung der Webseite vom 22.Juli 1999.

PGP - Pretty Good Privacy

PGP ist ein ursprünglich von Philip Zimmermann entwickeltes, weit verbreitetes und kostenfrei erhältliches kryptographisches Werkzeug, das im zusammen mit der Software verteilten Manual wie folgt charakterisiert wird: PGP combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of conventional cryptography, message digests for digital signatures, data compression before encryption, good ergonomic design, and sophisticated key management. And PGP performs the public-key functions faster than most other software implementations. PGP is public key cryptography for the masses. Zum Funktionsumfang gehören u.a. folgende Dienste:

- Digitales Signieren von Dokumenten und Nachrichten

Digitale Signaturen gestatten es dem Empfänger/Leser, mit hoher Sicherheit festzustellen, ob eine Datei, E-Mail etc. authentisch und integer ist, d.h. tatsächlich vom angegebenen Autor/Absender stammt und nicht durch Dritte erstellt bzw. von diesen nachträglich verändert wurde. Außerdem ist es für den Unterzeichner praktisch unmöglich, die geleistete Unterschrift später abzustreiten. Digitale Unterschriften ermöglichen also die Sicherung der Authentizität und Integrität von Nachrichten sowie einen Urhebernachweis.

- Verschlüsselung von per Computernetz transportierten Dokumenten und Nachrichten (E-Mails)

Computernetze wie z.B. das Internet sind als potentiell unsicher zu betrachten. Es besteht technisch die Möglichkeit, fremde Daten zu lesen und auch zu manipulieren. Durch eine Verschlüsselung unter Verwendung sog. symmetrischer Verfahren lässt sich die Vertraulichkeit der über das Netz ausgetauschten Informationen erreichen, d.h., nur die dazu berechtigten Empfänger gelangen in den Besitz der vertraulichen Information, da nur sie die verschlüsselte Nachricht entschlüsseln können.

Werden die betreffenden Dokumente zusätzlich digital signiert, so lassen sich neben der Vertraulichkeit auch noch die o.g. Eigenschaften (Authentizität, Integrität, Urhebernachweis) erreichen.

Ein wichtiges Anwendungsgebiet von PGP ist der kryptographische Schutz von E-Mails. Die Nachrichteninhalte werden dabei mit Hilfe des symmetrischen Algorithmus IDEA verschlüsselt. Die Verteilung der hierfür benötigten Schlüssel erfolgt mit Hilfe des asymmetrischen RSA-Verfahrens, das außerdem noch der Erstellung und Verifikation digitaler Signaturen dient. Der entscheidende Vorteil asymmetrischer gegenüber symmetrischen Verfahren besteht darin, dass der Schlüsselaustausch keinen sicheren Kanal erfordert, sondern problemlos über öffentliche Kanäle (Telefon, Internet, ...) erfolgen kann. Unter Verwendung von PGP ist es daher relativ leicht möglich, mit bisher unbekannten Personen oder Gruppen vertraulich zu kommunizieren.

- Schlüsselverwaltung

Die für die beiden zuvor genannten Funktionen benötigten Schlüsselpaare sowie die zu ihrer sicheren Weitergabe benötigten Zertifikate können durch PGP erzeugt und in sog. Schlüsselringen verwaltet werden.

- Konventionelle Kryptographie zum Schutz von Dateien im Dateisystem

Vertrauliche Informationen sollten auch im Dateisystem eines Computers verschlüsselt hinterlegt werden, da sonst generell die Gefahr besteht, dass sie in falsche Hände gelangen. Dies betrifft nicht nur netzbasierte Filesysteme (AFS, NFS, ...), wie sie z.B. in Rechenzentren typisch sind, sondern auch Festplatten im eigenen PC sowie Disketten, die entwendet oder leicht kopiert werden können, sofern andere Personen Zugang zu ihnen haben.

PGP gestattet deshalb den Schutz privater Datenbestände durch konventionelle Kryptographie, konkret durch den Algorithmus IDEA. Als externe Repräsentation der Schlüssel dienen vom Nutzer gewählte Passphrases. Dabei handelt es sich im Gegensatz zu Paßwörtern um beliebig lange Zeichenfolgen (z.B. ganze Sätze mit Leer- und Sonderzeichen). Um einen Mißbrauch in der Praxis auszuschließen, sollte man die Passphrases stets hinreichend lang und qualitativ so gut wählen, dass sie mit an Sicherheit grenzender Wahrscheinlichkeit nicht erraten werden können.

Unterstützte kryptographischer Verfahren

Die internationale Version von PGP basiert auf den folgenden drei kryptographischen Algorithmen:

- IDEA,
- RSA,
- MD5.

RSA und IDEA gelten gegenwärtig als sehr sicher, da es trotz intensiver Kryptoanalyse durch viele anerkannte Kryptologen aus dem akademischen und kommerziellen Umfeld nicht gelungen ist, diese Verfahren zu brechen. Zumindest liegen darüber keine Veröffentlichungen vor. Man muss dabei aber immer damit rechnen, dass es durchaus (z.B. bei den technisch und personell sehr gut ausgestatteten Geheimdiensten) der Fall sein kann, dass Verfahren existieren, die ein teilweises oder vollständiges Brechen dieser Kryptosysteme gestatten.

Davon wird die Öffentlichkeit nur in Ausnahmefällen erfahren, da sich das Mitteilungsbedürfnis der Geheimdienste und bestimmter staatlicher Behörden sehr stark in Grenzen hält. Ein Beispiel hierfür ist die Geschichte der Public-Key-Kryptographie. Als deren Erfinder werden Whitfield Diffie und Martin Hellman sowie unabhängig von beiden auch Ralph Merkle angesehen. Es gilt allerdings mittlerweile als ziemlich sicher, dass der NSA (National Security Agency der USA) und ähnlichen Organisationen die der Public-Key-Kryptographie zugrunde liegende Idee schon eher als den drei genannten Wissenschaftlern bekannt war.

Einige Informationen dazu wurden von Steve Bellovin auf der Seite The Prehistory of Public Key Cryptography [http://www.research.att.com/~smb/nsam-160/] zusammengetragen.

In den dort erwähnten, im Dezember 1997 veröffentlichten Papieren der CESG Communications-Electronics Security Group [URL: http://www.cesg.gov.uk/] schreibt der mittlerweile verstorbene CESG-Mitarbeiter James H. Ellis, dass er bereits in den 60er Jahren die Idee der Public Key Cryptography (PKC) entwickelt hatte, damals unter der Bezeichnung Non-Secret Encryption (NSE). Im Januar 1970 publizierte er im (geheimen) CESG-Report The Possibility of Secure Non-Secret Digital Encryption sein Existence Theorem, also den Nachweis, dass es NSE bzw. PKC überhaupt geben kann. Aus anderen CESG-Reports geht hervor, dass die CESG das Prinzip der bis heute breit genutzten asymmetrischen Verfahren Diffie-Hellman (1976) und RSA (1978) bereits einige Jahre vor deren Veröffentlichung in der akademischen Gemeinde kannte. Diese Papiere der CESG stehen unter der URL http://www.cesg.gov.uk/about/nsecret.htm öffentlich zur Verfügung.

Der Algorithmus MD5 wurde ursprünglich als ein sehr starkes Verfahren betrachtet. Neuere Untersuchungen durch den deutschen Kryptologen Prof. Hans Dobbertin haben aber gezeigt, dass Teile des Verfahrens erfolgreich attackiert werden können, so dass das Vertrauen in die Kollisionsresistenz des MD5 relativ stark gesunken ist. Zwar ist es bisher nicht gelungen, das gesamte Verfahren zu brechen, allerdings wird bei der Entwicklung neuer Protokolle, Verfahren und Werkzeuge der MD5 meist nicht mehr favorisiert, so wie das noch bis etwa 1995/96 der Fall war.

Für weitere Details sei auf die folgenden beiden Dokumente verwiesen:

- Dobbertins Artikel The Status of MD5 After a Recent Attack, der im Volume 2, No. 2 (Sommer 1996) der von der Firma RSA Data Securities, Inc. (RSADSI) herausgegebenen Zeitschrift CryptoBytes erschienen ist: http://www.rsa.com/rsalabs/pubs/cryptobytes/html/article_index.html.

- Bulletin Nr. 4 der RSA Laboratories vom 12. November 1996 mit dem Titel On Recent Results for MD2, MD4 and MD5. Es informiert über die bisher entdeckten Schwächen in den drei genannten Hashfunktionen, schätzt deren Folgen ab und gibt Empfehlungen für den Einsatz dieser Algorithmen. Das Dokument kann per WWW über die Seite http://www.rsa.com/rsalabs/html/bulletins.html bezogen werden.

IDEA, der International Data Encryption Algorithm, ist ein 1991 von Xuejia Lai und James L. Massey entwickeltes symmetrisches Verfahren mit einer festen Schlüssellänge von 128 Bit. Zur Ver- und Entschlüsselung einer Nachricht wird derselbe Schlüssel verwendet. Es sei darauf hingewiesen, dass IDEA patentiert ist und dass man für kommerzielle Anwendungen eine Lizenz benötigt.

PGP 2.6.x chiffriert generell alle Nutzerdaten mit IDEA, wobei der Schlüssel je nach Anwendungsmodus aus einer vom Anwender frei wählbaren Passphrase abgeleitet oder zufällig erzeugt wird. Vor der Verschlüsselung werden die Daten komprimiert. Dazu verwendet PGP Routinen von Mark Adler, Richard B. Wales und Jean-loup Gailly, die aus dem relativ bekannten Paket Info-ZIP stammen.

Das 1978 der Öffentlichkeit vorgestellte RSA-Verfahren gehört zur Klasse der asymmetrischen bzw. Public-Key-Verfahren. Die drei Buchstaben im Namen stehen für die drei Entwickler: Ronald L. Rivest, Adi Shamir und Leonard M. Adleman. Dabei spielt jeweils ein aus einem privaten und einem öffentlichen Schlüssel (Private Key und Public Key) bestehendes Schlüsselpaar eine Rolle. Lediglich der zur Entschlüsselung genutzte Private Key ist geheimzuhalten. Dagegen kann der zur Verschlüsselung verwendete Public Key bedenkenlos öffentlich bekanntgegeben werden.

Möchte man verschlüsselte Nachrichten austauschen, dann genügt es, die öffentlichen Schlüssel der jeweiligen Partner zu beschaffen. Die Gefahr des Abhörens besteht dabei nicht, da die Schlüssel ohnehin öffentlich sind. Lediglich ihre Manipulation muss zuverlässig verhindert werden. Dazu verwendet PGP sog. Zertifikate, die durch eine digitale Signatur die Echtheit eines öffentlichen Schlüssels, d.h. dessen tatsächliche Zugehörigkeit zu seinem vermeintlichen Eigentümer beurkunden.

PGP nutzt den relativ aufwendigen und daher zeitintensiven RSA-Algorithmus nicht zur Verschlüsselung der möglicherweise sehr umfangreichen Daten der Anwender, sondern nur zur Erstellung digitaler Signaturen sowie zur kryptographisch gesicherten Verteilung der zufällig generierten IDEA-Schlüssel, mit denen die Nutzerdaten chiffriert wurden.

Die digitalen Unterschriften werden nicht aus der Nachricht selbst, sondern aus dem mittels der Einweg-Hashfunktion MD5 (MD steht für Message Digest) gebildeten "Fingerabdruck" der Länge 128 Bit erzeugt. Das Verfahren MD5 stammt wiederum von Ronald Rivest und wurde 1992 im RFC 1321: The MD5 Message-Digest Algorithm beschrieben.

Unter einer Hashfunktion versteht man eine Funktion, die mindestens die folgenden zwei Eigenschaften besitzt:

1. Sie bildet Daten beliebiger Länge auf Werte fester Länge ab.
2. Der Funktionswert lässt sich aus dem Argument leicht (d.h. mit wenig Aufwand) berechnen.

Für die Kryptographie sind in der Regel Einweg-Hashfunktionen von Interesse, also Hashfunktionen, die sich nur äußerst schwer, d.h. mit extremem Aufwand (der möglichst die gesamte verfügbare Rechen- und Speicherkapazität bei weitem übersteigt) invertieren lassen. Einweg-Hashfunktionen zeichnen sich also durch die beiden folgenden zusätzlichen Eigenschaften aus:

1. Es ist rechnerisch nicht möglich, eine Nachricht zu ermitteln, die einen vorgegebenen Hashwert besitzt.
2. Es ist rechnerisch ebenfalls nicht möglich, zu einer gegebenen Nachricht eine zweite, davon verschiedene Nachricht zu konstruieren, die denselben Hashwert wie die erste hat.

Für verschiedene Anwendungsfälle werden kollisionsresistente Einweg-Hashfunktionen gefordert. Darunter versteht man jene Einweg-Hashfunktionen, die als zusätzliche Eigenschaft die Kollisionsresistenz aufweisen. Sie besagt, dass es rechnerisch unmöglich ist, eine Kollision, d.h. zwei unterschiedliche, frei wählbare Nachrichten zu erzeugen, die

denselben Hashwert haben, auch wenn es in der Praxis meist unendlich viele derartige Nachrichten gibt.

In seinem Artikel The Status of MD5 After a Recent Attack sagt Dobbertin, dass die von ihm präsentierten Attacken für praktische Anwendungen des MD5 zwar noch keine Gefährdung darstellen, einer solchen aber schon recht nahekommen. Deshalb empfiehlt er, den MD5 künftig nicht mehr in Applikationen einzusetzen, die kollisionsresistente Einweg-Hashfunktionen benötigen, wozu typischerweise Verfahren zur Erstellung digitaler Signaturen zählen. Als Alternativen nennt er SHA-1 und RIPEMD-160. Diese beiden Verfahren werden allerdings von 2.6.x nicht unterstützt. Bei PGP 5.x/6.x kommt dagegen vorzugsweise SHA-1 zur Anwendung.

Wie das folgende Zitat aus einem News-Artikel zeigt, vertritt Markus G. Kuhn die Ansicht, dass die Kollisionsresistenz von Hashfunktionen bei digitalen Signaturen entbehrlich ist:

```
Dass man fuer digitale Unterschriften Kollisionsresistenz einer Hashfunktion braucht halte ich fuer ein weit verbreitetes Missverstaendnis. Die Einwegeigenschaft sollte ausreichen wenn man gesetzlich alle Hash preimages als unterschrieben ansieht. Nur der Unterschreiber kann Kollisionen ausnutzen, also muss das nur zu seinem Nachteil ausgelegt werden und schon braucht man keine Kollisionsresistenz mehr. Man muss dann nur sicherstellen, dass man keinen Text unterschreibt den man nicht selbst erstellt hat, aber das laesst sich ja durch Hinzufuegen von ein paar Zufallsbits an den Anfang eines Textes bevor man unterschreibt vermeiden.

Aus dem gleichen Grund reicht es voellig, bei PGP nur die ersten 10 Bytes des Fingerprints zu vergleichen. Man kann so Platz auf Visitenkarten sparen ohne Sicherheit einzubuessen.
```

Es empfiehlt sich natürlich generell, regelmäßig und aufmerksam die neusten kryptographischen Erkenntnisse zu verfolgen, um so früh wie möglich auf erfolgreiche Attacken gegen die eingesetzten kryptographischen Verfahren reagieren zu können.

PGP 5i, die internationale Freeware-Version von PGP 5.x, bietet eine hohe Kompatibilität mit älteren PGP-Versionen:

- PGP 5i kann Nachrichten, Schlüssel und Signaturen von PGP 2.x verarbeiten, sofern die RSA-Schlüssel nicht länger als 4096 Bits sind.
- PGP 5i kann Nachrichten, Schlüssel und Signaturen erzeugen, die durch PGP 2.6.x, nicht aber durch ältere Versionen (z.B. PGP 2.3a) verarbeitet werden können (natürlich nur, sofern man RSA-Keys verwendet).
- Fazit: Bei ausschließlicher Verwendung von RSA-Keys ist PGP 5i mit allen heute gängigen PGP-Versionen kompatibel. DSA- und ElGamal-Keys sollte man aber nur verwenden, wenn der Partner mindestens über PGP 5.0 verfügt.

PGP 5.x/6.x unterstützt bei den internationalen und einigen anderen Versionen weiterhin die Algorithmen IDEA, RSA und MD5, um auch Nachrichten und Zertifikate verarbeiten zu

können, die mit älteren PGP-Versionen erstellt wurden. Vorzugsweise werden aber andere kryptographische Verfahren verwendet, bei deren Einsatz folglich PGP-Dateien entstehen, mit denen die älteren Versionen nicht umgehen können.

Das RSA-Verfahren wird dabei durch zwei verschiedene asymmetrische Algorithmen ersetzt, die jeweils ein separates Paar aus öffentlichem und privatem Schlüssel verwenden:

- DSA für das Erstellen digitaler Signaturen und

- ElGamal für das Verschlüsseln mit Hilfe eines Public-Key-Kryptosystems.

Für das digitale Signieren und das Chiffrieren von Dokumenten werden hier also getrennte Schlüsselpaare verwendet, wogegen bei RSA ein und dasselbe Schlüsselpaar beide Funktionen erfüllt. Würde jemand z.B. durch staatliche Behörden gezwungen, seinen geheimen RSA-Key offenzulegen, um den Behörden das Dechiffrieren von Nachrichten mit vermeintlich kriminellem Inhalt zu ermöglichen, wären diese Stellen automatisch in der Lage, beliebige Dokumente im Namen der betreffenden Person digital zu signieren, wodurch sämtliche Signaturen, die mit dem offengelegten Schlüssel erstellt wurden bzw. werden, ihren Sinn schlagartig komplett verlieren würden.

Bei der Trennung der beiden Schlüsselpaare ist dies nicht der Fall. Die Behörden können mit dem ElGamal-Key zwar Nachrichten dechiffrieren, nicht aber Unterschriften fälschen, solange sie nicht im Besitz des DSA-Keys sind.

LITERATUR- UND QUELLENHINWEISE

[Aho]
Aho, A.V., Sethi, R. und Ullmann, J. D.
Compilerbau. In zwei Teilen.
Addison-Wesley, Bonn 1988

[Ahrweiler]
Ahrweiler, Chris
Kryptologie - Verschlüsselung im Informationszeitalter
Skript zum Workshop am 28. Oktober 1996
Internet, URL: http://www.docjojo.com/ [Abruf 1999-09-14]

[Bachhuber]
Bachhuber, W.
Einführung in die Kryptologie.
Internet, URL: http://www.bachhuber.com/Security/Cryptologie.html, Abruf 1999-09-08.

[BaeNieSchr]
Baeumle-Courth, P., Nieland, S. und Schröder, H.
Wirtschaftsinformatik
Oldenbourg-Verlag, München 2004

[Baeumle1]
Baeumle, P.
C++ kompakt
S+W Steuer- und Wirtschaftsverlag, Hamburg 1996

[Baeumle2]
Baeumle, P. und Alenfelder H.
Compilerbau
S+W Steuer- und Wirtschaftsverlag, Hamburg 1995

[Baeumle3]
Baeumle-Courth, P.
C++ kompakt - Eine Einführung in C++ (5 Folgen)
in: about*IT* (Internetbasiertes Magazin für die IT-Branche)
Internet, URL: http://www.aboutIT.de/00/07/02.html, Abruf 2004-06-15.

[Balke]
Balke, L. und Böhling, K. H.
Einführung in die Automatentheorie und die Theorie formaler Sprachen
Bibliographisches Institut, Mannheim 1993

[Beedgen]
Beedgen, R.
„Elemente" der Informatik.
Ausgewählte mathematische Grundlagen für Informatiker und Wirtschaftsinformatiker
vieweg Verlag, Braunschweig / Wiesbaden 1993

[Bengel]
Bengel, G.
Betriebssysteme. Aufbau, Architektur und Realisierung.
Hüthig Verlag, Heidelberg 1990

[Beutelspacher]
Beutelspacher, A.
Kryptologie.
Vieweg Verlag, Braunschweig/Wiesbaden 1996

[BeuSchWol]
Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.
Moderne Verfahren der Kryptographie - Von RSA zu Zero-Knowledge
Verlag Vieweg, Braunschweig/Wiesbaden 1998 (2.Auflage)

[Born]
Born, G.
Referenzhandbuch Dateiformate
Addison-Wesley, Bonn, 1994 (3. Auflage)

[CERN]
Online-Quelle beim Centre Européen de Recherche Nucléaire (CERN):
http://www.cern.ch/CERN/WorldWideWeb/WWWandCERN.html

[Codd]
Codd, E. F.
A relationship model of data for large shared data banks
CACM 13, Juni 1970

[Dewdney]
Dewdney, A. K.
Der (neue) Turing Omnibus
Eine Reise durch die Informatik mit 66 Stationen
Springer Verlag, Heidelberg 1995

[DifHel76]
Diffie, W. und Hellman, M.
New Directions in Cryptography
Transactions of the IEEE - Information Theory, Vol. 6 (1976), pp 644-654

[DINO]
Online-Quelle bei DINO-Online:
http://www.dino-online.de/internet.html

[Dworatschek]
Dworatschek, S.
Grundlagen der Datenverarbeitung
Walter de Gruyter, Berlin 1977 (6.Auflage)

[EbbFluTho]
Ebbinghaus, H.-D., Flum, J. und Thomas, W.
Einführung in die mathematische Logik
Darmstadt, 1978

[Ernst]
Ernst, H.
Grundkurs Informatik
Wiesbaden, 3.Auflage 2003

[Fischer]
Fischer, J., Dangelmaier, W., Herold, W., Nastansky, L. und Wolff, R.
Bausteine der Wirtschaftsinformatik
S+W Steuer- und Wirtschaftsverlag, Hamburg 1995

[Hansen]
Hansen, H. R.
Wirtschaftsinformatik I
Gustav Fischer / UTB, Stuttgart 1983 (ff)

[Hughes]
Hughes, John G.
Objektorientierte Datenbanken
Hanser Verlag, München 1991

[Landauer]
Landauer, Thomas
Überblick über die Arbeitsweise von PGP
Internet, URL: http://www.unet.univie.ac.at/~a9204810/PGP.htm, letzte Änderung 25.04.1999
[Abruf vom 1999-09-14]

[Lehner]
Lehner, F., Hildebrand, K. und Maier, R.
Wirtschaftsinformatik. Theoretische Grundlagen.
Hanser Verlag, München 1995

[Matzdorff]
Matzdorff, K.
Objektorientierte Softwareentwicklung mit C++
S+W Steuer- und Wirtschaftsverlag, Hamburg 1994

[Misgeld]
Misgeld, Wolfgang
SQL - Einstieg und Anwendung
Hanser Verlag, München 1991

[Nagl]
Nagl, W. D.
Computertechnologie und Managementpraxis
Addison-Wesley, Bonn 1992

[NivZuc]
Niven, I. und Zuckerman, H. S.
Einführung in die Zahlentheorie I
BI Hochschultaschenbücher, Mannheim 1976

[Ortmann]
Ortmann, D.
Access für Windows 95 für Datenbankentwickler
Hanser Verlag, München 1996 (2.Auflage)

[Sauer]
Sauer, Hermann
Relationale Datenbanken - Theorie und Praxis
Addison-Wesley Verlag, Bonn/München 1991

[Schöning]
Schöning, U.
Logik für Informatiker
Spektrum - Akademischer Verlag, Heidelberg 1995

[SchwSör]
Schwarz, J. und Sörmann, G.
Kompressionsalgorithmen
Worms, 1995/1996
Im Internet unter http://www.ztt.fh-worms.de/de/sem/ws95_96/kompressionsalgorithmen/
Kompressionsalgorithmen.html abrufbar.

[Schwinn]
Schwinn, Hans
Relationale Datenbanksysteme
Hanser Verlag, München 1992

[Stahlknecht]
Stahlknecht, P.
Einführung in die Wirtschaftsinformatik
10.Auflage, Springer Lehrbuch, Heidelberg 2002

[Tolksdorf]
Tolksdorf, R.
Internet - Aufbau und Dienste
International Thompson Publishing, Bonn 1997

[Trapp]
Trapp, H.
Pretty Good Privacy - Überblick über die Arbeitsweise von PGP
http://www.tu-chemnitz.de/~hot/pgp.html, letzte Änderung 22.Juli 1999 [Abruf 1999-09-11].

[Vetter]
Vetter, M.
Aufbau betrieblicher Informationssysteme mittels konzeptioneller Datenmodellierung
B.G. Teubner Verlag, Stuttgart, Reihe Leitfäden der angewandten Informatik, 2.Auflage 1985

[Vossen]
Vossen, G. und Witt, K.-U.
Das SQL/DS-Handbuch
Addison-Wesley Verlag, Bonn/München 1988

[W3C]
Online-Quelle beim World Wide Web Consortium (W3C):
http://www.w3.org/

[Welzel]
Welzel, P.
Computervernetzung. Verteilte Verarbeitung in offenen Systemen.
S+W Steuer- und Wirtschaftsverlag, Hamburg 1995

[Winkler]
Winkler, B. L.
Grundlagen der digitalen Bild- und Tonverarbeitung
http://www.rz.uni-bayreuth.de/lehre/dibito/vorlesung/dibito.html
Lokale Spiegelung im FHDW-Intranet: www.bg.bib.de/FHDW/Intranet/gdi/cod/DigitaleBildTonVerarbeitung.pdf

[Wirth]
Wirth, N.
Algorithmen und Datenstrukturen
Teubner Verlag, Stuttgart 1975 (ff)

[Wirth2]
Wirth, N.
Compilerbau
Teubner Verlag, Stuttgart 1986

S. Stichwortverzeichnis

D

E

O

P

Q

R

S

T

U

V

W

X

Z

Informatik-Studium

Grundlagen der Informatik